UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

PROYECTO FIN DE CARRERA

Ingeniería Industrial

SISTEMA DE SEGURIDAD BASADO EN EL CÁLCULO DE LAS TRAYECTORIAS DE LOS VEHÍCULOS MEDIANTE CÁMARA EMBARCADA

AUTOR: Luis Alberto Aranda Barjola

TUTOR: Arturo de la Escalera Hueso

OCTUBRE 2012

Título: Sistema de seguridad basado en el cálculo de las trayectorias de los vehículos mediante cámara embarcada
Autor: Luis Alberto Aranda Barjola
Tutor: Arturo de la Escalera Hueso

TRIBUNAL

Presidente: José María Armingol Moreno

Secretario: Fernando García Fernández

Vocal: Guillermo Carpintero del Barrio

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 8 de Octubre de 2012 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de ___

PRESIDENTE

SECRETARIO VOCAL

Agradecimientos

Primero agradecer a mis padres, con su apoyo, cariño y confianza, han logrado que realice dos de mis más grandes metas en la vida: la culminación de esta etapa de aprendizaje y el hacerlos sentirse orgullosos de mi persona.

A los profesores de la universidad, y en concreto a mi tutor, sin ellos este proyecto no habría sido llevado a cabo.

A todos y cada uno de mis compañeros de universidad, gracias por la inestimable ayuda y por los grandes momentos que hemos pasado; espero sigamos juntos en esta nueva etapa que comienza, pero no como compañeros, sino como verdaderos amigos.

En especial a mi otra mitad, por su comprensión, amor y ternura; un tesoro que también encontré en esta carrera.

Gracias a todos

Índice general

Resumen

En el presente proyecto se explica el desarrollo de una aplicación informática que realiza el procesado de una secuencia de imágenes captadas por una cámara estéreo embarcada en un turismo. Este software permite detectar vehículos que circulen en el mismo sentido que el turismo, determina a qué distancia se encuentran y su velocidad de circulación; asimismo efectúa una búsqueda de los vehículos detectados en imágenes previas para proceder a su seguimiento.

Capítulo 1

Introducción

En los últimos años el campo de investigación de la Visión Artificial o Visión por Computador ha experimentado un crecimiento exponencial debido sobre todo al aumento de la capacidad de procesamiento y de la potencia de los microprocesadores actuales; sin embargo, la visión artificial es una ciencia relativamente moderna, pues no fue hasta 1961 cuando, en el MIT (*Massachusetts Institute of Technology*), Larry Roberts junto a un grupo de investigadores desarrolla el primer programa que capta imágenes a través de una cámara y procesa esta información para "comprender" lo que estas imágenes representan. Una primera aplicación que llevaron a cabo fue el "mundo de los micro-bloques" [9], en el cual un robot era capaz de percibir un conjunto de bloques sobre una mesa, moverlos y apilarlos.

Desde entonces la Visión por Computador no ha parado de crecer y se ha expandido por diversas ramas tecnológicas, destacamos:

- *Biología*: se puede hacer una distinción entre aplicaciones microscópicas y macroscópicas. A nivel microscópico existen trabajos basados en técnicas de segmentación que permiten aislar los organismos presentes en una imagen para su identificación o para determinar el número de ellos. Por otro lado, en imágenes macroscópicas la Visión Artificial puede ayudar a reconocer texturas y colores de distintas especies vegetales o cuantificar su crecimiento mediante diferencia de imágenes.

- *Geología*: la técnica de diferencia, o resta de imágenes, comentada previamente, permite detectar movimientos entre dos imágenes consecutivas separadas un cierto intervalo temporal, por lo que se podría observar la variación del terreno con el tiempo. Otra aplicación a destacar en esta área es la reconstrucción tridimensional de zonas mediante visión estereoscópica.

- *Meteorología*: las mismas técnicas de diferencia de imágenes, junto con otras de predicción del movimiento, permiten estimar la evolución de masas nubosas o cualquier otro fenómeno meteorológico a través de imágenes recibidas vía satélite.

- *Medicina*: dentro de este vasto campo de estudio existen multitud de aplicaciones médicas de gran utilidad, por ello, se distinguen las siguientes sin ser demasiado exhaustivos: coloreado de regiones de interés en una radiografía mediante técnicas de pseudocolor, obtención del entramado de vasos capilares, o nervios de un tejido mediante la extracción de bordes, o la distinción entre tejidos sanos y cancerígenos por el color, entre otras.

- *Industria*: la Visión por Computador ha aportado muchas ventajas a este sector, la más importante a recalcar es la reducción de la complejidad y el aumento de la eficiencia en los procesos de inspección y control de calidad, ya que permite clasificar objetos en función de su tamaño, verificar que estas dimensiones sean correctas, que el acabado superficial sea el adecuado, etc. Se consigue evitar así que en la manufactura en cadena se genere una larga serie de productos con defectos que aumentarían los costes y reducirían la competitividad de la empresa.

Como podemos observar, la diversidad de campos de aplicación de la Visión Artificial es grande, sólo lo limita la imaginación y la tecnología presente.

Es notable destacar que, tanto las ramas aquí comentadas como las no citadas tienen un elemento en común: la cámara digital.

Las cámaras digitales más usadas en la Visión por Computador son las CCD/CMOS y monocular/estereoscópicas.

- *CCD/CMOS*: las cámaras CCD [26] o de dispositivo de carga acoplada (*charge-coupled device*) constan de un chip de silicio cuya superficie se divide en diminutas células fotoeléctricas o píxeles sensibles a la luz. Cuando un fotón (partícula de luz) alcanza un píxel, se contabiliza una pequeña carga eléctrica que permite registrar la imagen. Posteriormente, esta imagen es procesada por la cámara y almacenada en la tarjeta de memoria.

  Las cámaras CMOS (*complementary metal-oxide-semiconductor*) se denominan así porque emplean tecnología CMOS para controlar los fotodiodos que captan la luz, es decir, su sensor, al igual que en las cámaras CCD, está basado en el efecto fotoeléctrico, pero utiliza componentes electrónicos CMOS (transistores MOSFET) para controlar estos fotositos.

- *Monocular/estereoscópicas*: las cámaras estereoscópicas [25] o cámaras 3D reciben su nombre de la visión estereoscópica humana. Estas cámaras son capaces de capturar imágenes tridimensionales, para ello se basan en la visión binocular humana, donde dos imágenes (una para cada ojo) se mezclan en el cerebro creando una única imagen en tres dimensiones. Este comportamiento se logra imitar utilizando una cámara con dos objetivos o dos cámaras monoculares separadas una cierta distancia entre sí que captan la misma imagen en el mismo instante pero desde puntos espaciales distintos, consiguiendo como resultado una imagen tridimensional.

Entre las dos primeras, las más populares en aplicaciones profesionales y en cámaras digitales son las de tecnología CCD pues, aunque presentan mayor consumo eléctrico y, por tanto, menor autonomía, no presentan el elevado ruido de patrón fijo de las CMOS. Las imágenes estereoscópicas, en la mayoría de los casos, se consiguen reproducir utilizando dos cámaras separadas estratégicamente, pues el proceso de sincronización y posicionamiento no es excesivamente complejo, y permite una mayor flexibilidad en cuanto a ángulo y colocación de cámaras, pudiendo obtener diversos resultados.

El software implementado en este proyecto está diseñado para cámaras estereoscópicas, pues se ha decidido desarrollar específicamente para el vehículo de pruebas de la Universidad Carlos III de Madrid, el IVVI 2.0. [22]

Figura 1.1: Vehículo IVVI de la Universidad Carlos III de Madrid

El IVVI o Vehículo Inteligente basado en Información Visual, es una plataforma de experimentación para el desarrollo de Sistemas de Ayuda a la Conducción mediante el Análisis de Imágenes y la Visión por Computador. En este vehículo se están implementando diversos trabajos que toman como base una secuencia de imágenes, lo que permite analizar una gran diversidad de situaciones y probar los algoritmos ante casos reales. El 2.0 indica que es el segundo vehículo al que se le han instalado los equipos necesarios para el desarrollo de Sistemas de Ayuda a la Conducción.

Figura 1.2: Cámara de barrido progresivo HITACHI

Este vehículo dispone de un sistema estéreo blanco y negro formado por dos cámaras de barrido progresivo para poder capturar imágenes en movimiento y evitar los problemas inherentes al vídeo entrelazado (ver figura 1.3). Este sistema es el proveedor de las imágenes que necesita la aplicación explicada en el presente proyecto.

Figura 1.3: Diferencias entre barrido progresivo y barrido entrelazado [1]

De igual forma el IVVI tiene instalada una cámara a color para la localización e identificación de señales de tráfico, así como dos cámaras de infrarrojo lejano que captan el calor que desprenden los objetos, con las que se pueden percibir obstáculos como peatones u otros vehículos en condiciones de visibilidad adversa.

Figura 1.4: Ejemplo reconocimiento de señales de tráfico

También se percibe el interior del vehículo con una cámara que enfoca al rostro del conductor para evaluar en todo momento su grado de atención. Una iluminación infrarroja permite trabajar durante la noche sin molestar a los ocupantes del coche.

Para conocer el estado del vehículo se dispone de una sonda CAN-Bus que obtiene información del funcionamiento del vehículo así como de un sistema GPS-IMU que da la información de la posición y velocidad del vehículo, con ello se logra una percepción continua del entorno y se puede reaccionar en cualquier momento ante un peligro.

La alimentación eléctrica necesaria para el funcionamiento de los distintos equipos y sensores se obtiene de la batería a través de un convertidor DC/AC directamente conectado a ella. Por último, tres ordenadores colocados en el maletero del vehículo se encargan del procesamiento de los sistemas de Visión por Computador, estos computadores pueden ser manejados simultáneamente por un operador, situado en el asiento trasero, gracias a un multiplexador para las señales de vídeo, ratón y teclado.

Todas estas aplicaciones convergen en el objetivo del IVVI: mejorar la seguridad al volante y minimizar, en lo posible, los errores humanos.

Este objetivo no sólo lo persigue el proyecto IVVI comentado anteriormente, sino también otros como el recientemente publicado trabajo de Volvo. [28]

Figura 1.5: Volvo V40

Los ingenieros de Volvo han desarrollado sistemas de Visión Artificial y un sistema radar que implementarán en los nuevos Volvo V40 para ayudar a reducir atropellos. El sistema consiste en un radar integrado en la parrilla delantera del coche, una cámara ubicada en el espejo central y una unidad de control. El radar se encarga de detectar peatones o vehículos delante del coche y determina la distancia, mientras que la cámara determina el tipo de objeto.

Este sistema, en caso de emergencia, envía una alerta sonora combinada con una señal de luz en la luna delantera, si el conductor no reacciona a la señal y la colisión es inminente, los frenos se accionan automáticamente. Así, según Volvo, se pueden evitar colisiones con peatones a una velocidad de hasta 35km/h si el conductor no reacciona a tiempo.

Otro trabajo relacionado con el ámbito de la circulación de vehículos es el de las cámaras *BirdWatch RL (Red Light)*, implementadas para detectar automóviles que se saltan semáforos en rojo [21]. Este dispositivo, de reducidas dimensiones, integra todo lo necesario para garantizar su correcto funcionamiento: una cámara para grabar el paso del vehículo y su matrícula, iluminación para poder trabajar de noche, un procesador donde se analiza la información y se toman las decisiones correspondientes, y comunicación para el envío de los avisos o notificaciones.

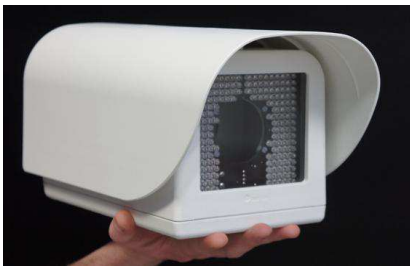

Figura 1.6: Sistema *BirdWatch RL*

Aunque el uso originario del sistema *BirdWatch RL* es el control de los vehículos que se saltan semáforos en rojo, dada su sencillez de uso y su facilidad de implantación este sistema se está volviendo más versátil y se le están incorporando nuevas funcionalidades como la información en tiempo real del tráfico, lo que permite el estudio de las redes viarias, su capacidad y nivel de circulación, densidad del tráfico, accidentes, etc.

Centrándonos en el tema que nos concierne, el objetivo del proyecto es el desarrollo de una aplicación software que, combinada con un conjunto de elementos hardware de procesamiento, control y adquisición de datos como el sistema estéreo de barrido progresivo anteriormente comentado, ayude en la conducción de un vehículo dotándolo de cierta "inteligencia".

Esta aplicación informática tiene una serie de cometidos, el primero, y más importante de todos, es el procesamiento de la imagen obtenida del sistema de cámaras para hallar los vehículos presentes en la misma. Este proceso se realiza mediante las denominadas Cascadas de Haar o Clasificadores de Haar [2] [6] [11] [24] que se explicarán posteriormente.

Una vez determinado el o los vehículos que circulan en el mismo sentido que el turismo dotado de sistemas de Visión por Computador, se procede al almacenaje en memoria de los vehículos detectados para poder buscar su posición en la siguiente imagen. Para ello, se extraen pequeñas regiones de la imagen [17] conteniendo un trozo de la zona trasera de cada vehículo detectado. Estos fragmentos son procesados y convertidos en números. Luego se buscará en la siguiente imagen los números más próximos a ellos en las zonas más probables de la imagen (pues el movimiento de un automóvil es continuo en el espacio, no puede ir "dando saltos" por la imagen) consiguiendo así un seguimiento de los vehículos detectados.

Por último, se hará una estimación de la distancia entre los vehículos encontrados y el portador del sistema de Visión Artificial mediante los denominados Mapas de Disparidad [2] [16] y, consecuentemente, se dará un valor aproximado de la velocidad relativa de los vehículos a partir de parámetros espaciales y temporales.

El presente texto se estructura en nueve secciones. El capítulo 1 es, como el lector ha podido comprobar, una breve introducción al mundo de la Visión por Computador y sus aplicaciones, así como una perspectiva a grandes rasgos de la finalidad de este proyecto.

En el capítulo 2 se explicarán las bases teóricas y herramientas fundamentales para la comprensión del proyecto, sin las cuales no se habría podido llevar a cabo.

En el capítulo 3 se incluyen los pasos para la creación o entrenamiento de una Cascada de Clasificadores de Haar que permiten detectar cualquier objeto rígido que tenga vistas distinguibles [2], como pueden ser rostros humanos o, según nuestro caso, coches.

Gracias a las Cascadas de Haar comentadas en el capítulo 3 se puede llevar a cabo la detección de vehículos. En el capítulo 4 se estudiarán dos métodos: la detección de la parte trasera de un vehículo mediante una única cascada, y la detección de un vehículo por partes agrupando los resultados de varias cascadas especializadas en encontrar regiones concretas del vehículo como las ruedas, las esquinas superiores y la matrícula. Al final del capítulo se explica cómo se dotó al programa de memoria para hacerlo más "inteligente", consiguiendo así mejores resultados pues "recuerda" cuantos vehículos había antes y su posición en la imagen.

En el capítulo 5, y a partir de lo logrado en el capítulo anterior, se detalla cómo extraer un patrón identificativo del vehículo detectado y cómo buscarlo en la siguiente imagen de la secuencia, consiguiendo así el seguimiento del mismo. Para ello se usará un parámetro estadístico también explicado en esta sección: el rango intercuartílico.

En el capítulo 6 se muestra cómo obtener dos parámetros muy interesantes a la hora de realizar el control del vehículo: la distancia entre los vehículos detectados y el turismo dotado de Visión Artificial, y su velocidad. Ambas medidas pueden proporcionar muchas aplicaciones tanto para la creación de transportes autónomos como para la implementación de mayor seguridad en vehículos.

En el capítulo 7 se presentan los resultados y las conclusiones del proyecto, mientras que algunas de las posibles aplicaciones de este trabajo pueden encontrarse en el capítulo 8.

Finalmente, en el capítulo 9 se estima el presupuesto necesario para el desarrollo del presente proyecto.

Capítulo 2

Base Teórica

No se puede comenzar a hablar de un proyecto sin explicar antes su base teórica, ya que ésta es la plataforma que lo sustenta y sobre la cual se diseña el estudio. En nuestro proyecto este capítulo está dedicado a la Visión por Computador, así como a las herramientas informáticas y lenguajes de programación que permitieron su desarrollo. Por último, al final de esta sección se hace una introducción a los Clasificadores de Haar, necesaria para comprender el siguiente capítulo y una de las partes fundamentales de este proyecto.

2.1 Visión por Computador

En el capítulo anterior usábamos indistintamente los términos Visión por Computador y Visión Artificial, esto no es del todo exacto, por lo que haremos una breve distinción.

Denominamos Visión por Computador al conjunto de técnicas para obtener información útil de una o varias imágenes por medio de un dispositivo electrónico programable. Es decir, el objetivo de la Visión por Computador es obtener y analizar la información contenida en la imagen, en base a la extracción de sus características más representativas.

Por otro lado, la Visión Artificial se define como el campo de la Inteligencia Artificial que, mediante la utilización de técnicas adecuadas, permite adquirir, procesar y analizar cualquier tipo de información obtenida a través de imágenes digitales. Es decir, la Visión Artificial es la aplicación de la Visión por Computador a la industria con el objetivo de controlar un sistema automático usando información visual.

También, en el capítulo 1 y en otras partes de esta memoria se utiliza el término "procesamiento de una imagen", por lo que es conveniente explicarlo brevemente aquí. Este concepto es muy sencillo de entender, cuando decimos que estamos *procesando una imagen* nos referimos a que le estamos aplicando un conjunto de técnicas con el objetivo de mejorar su calidad o facilitar la búsqueda de información, en otras palabras, estamos transformando la imagen inicial en otra u otras similares a la inicial pero con algunas características modificadas. Un ejemplo sería la manipulación del contraste o el brillo, la aplicación de filtros para la eliminación de ruido, etc.

Volviendo al tema de la Visión por Computador, y una vez comprendidos estos conceptos, se pueden exponer los pasos de este tipo de proyectos.

Figura 2.1: Etapas de un sistema de Visión por Computador

En la figura 2.1 se pueden ver las fases que componen un proyecto de Visión por Computador junto a un ejemplo ilustrativo de un sistema que, mediante la lectura de la matrícula y posterior búsqueda en una base de datos, dispone a elección del operario el dejar entrar o salir al vehículo accionando la barrera que le impide el paso [12]. Para ello le informa si esa matrícula está o no en la base de datos.

El proceso parte del objeto a detectar, el cual es captado por un sistema de cámaras de muy diversos tipos (tal y como se comentó en el capítulo 1). En el interior de estas cámaras es donde se forma la imagen que, posteriormente, será preprocesada para mejorar su calidad en caso de ser necesario.

Una vez adquirida la imagen se procede a su análisis mediante la extracción de las características más relevantes o, simplemente, aquellas que necesitemos para reconocer y localizar el objeto captado.

Con la información obtenida se interpreta la imagen y, finalmente, se lleva a cabo la toma de decisiones.

Cabe destacar que la extracción de características es una de las fases críticas en el proceso, pues de ella depende la correcta o incorrecta actuación final. Este paso ha sido estudiado por gran cantidad de investigadores, llegándose a desarrollar diversos métodos pero, habitualmente, todos ellos constan de una detección de contornos seguida de una división de la imagen en varias regiones u objetos, proceso conocido como segmentación.

En nuestro proyecto la extracción de características se consigue gracias a los Clasificadores de Haar comentados al final de este capítulo.

2.2 Lenguajes de Programación C y C++

El software desarrollado en este proyecto está escrito en lenguaje de programación C++ [8], por lo que es necesario hacer una breve introducción al mismo, así como a su predecesor, el lenguaje C, ya que C++ es una extensión de él.

A partir de las investigaciones de Martin Richards sobre el lenguaje de programación BCPL (*Basic Combined Programming Language*), surge otro llamado B e inventado por Ken Thompson, este lenguaje fue influencia directa en la aparición del lenguaje C.

El lenguaje C nace en los Laboratorios Bell de la empresa AT&T entre los años 1970 y 1972 gracias a Brian Kernighan y Dennis Ritchie, quienes lo desarrollaron para escribir el código del Sistema Operativo UNIX. Posteriormente, en 1978, Kernighan y Ritchie publican el libro "*The C Programming Language*" donde se detallan las características de este lenguaje.

A mediados de los ochenta, C ya se había hecho muy popular y había sido aceptado por la mayoría de los programadores, por lo que aparecieron en el mercado numerosos compiladores C y aplicaciones que habían sido rescritas a él para aprovechar sus ventajas. Durante este periodo los fabricantes introducen algunos cambios y mejoras en el lenguaje, lo que llevó a su estandarización ANSI, donde se establecieron las especificaciones de lo que hoy en día se conoce como ANSI C.

El lenguaje C, según la jerarquía de lenguajes, se encuentra en un nivel intermedio entre Pascal y Ensamblador, en otras palabras, pretende ser un lenguaje de alto nivel con la versatilidad del bajo nivel. Inicialmente fue creado para la programación de Sistemas Operativos, Intérpretes, Editores, Ensambladores, Compiladores y Administradores de Bases de Datos, pero actualmente se puede utilizar para desarrollar todo tipo de programas.

Su principal característica es que es un lenguaje portable, es decir, las aplicaciones escritas para un tipo de computadora se pueden adaptar para que funcionen en otras distintas. También es estructurado, pues se divide en módulos independientes entre sí.

Cuando desarrollaban el lenguaje, Kernighan y Ritchie decidieron seguir una política en la que la máxima era la creación de aplicaciones sin que hiciera falta escribir mucho texto. Para lograrlo, redujeron el número de palabras clave, consiguiendo así compiladores pequeños y eficientes que permitirían aumentar la productividad por día de los programadores.

Otra característica a destacar es que C se apoya en funciones de librería para realizar instrucciones complejas, como la entrada/salida de información, el manejo de cadenas de caracteres, etc.

Estas características, y otras más que no se han mencionado, hacen de C un lenguaje rápido de aprender, que deriva en compiladores sencillos de diseñar, robustos, y que generan objetos pequeños y eficientes. Por ello, es un lenguaje adecuado para la programación de sistemas, aplicaciones científicas, sistemas de Bases de Datos, software gráfico o análisis numérico.

Aún así, la excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente, no se detectan a simple vista. A pesar de todo, C ha demostrado ser un lenguaje extremadamente eficaz y expresivo que ha influido a otros como Java, PHP, o el propio C++.

Como se dijo al inicio de este epígrafe, el código del proyecto está escrito en C++. Este lenguaje de programación fue diseñado por Bjarne Stroustrup entre 1983 y 1985. Bjarne Stroustrup trabajó con los Laboratorios Bell de AT&T durante varios años para desarrollar y mejorar C++. El objetivo que pretendía conseguir era extender el ya célebre lenguaje de programación C adicionándole propiedades que permitieran la manipulación de objetos.

Como se dijo, C++ deriva de C pero, en realidad, es un superconjunto de C, pues nació para añadirle cualidades y características de las que carecía. Es por ello que el lenguaje C++ tiene una compatibilidad elevada con éste, principalmente porque comparte gran cantidad de código C que ha sido reutilizado, pero también porque se buscaba facilitar el paso de los programadores de C al nuevo lenguaje C++.

El resultado es que, como su ancestro, sigue muy ligado al hardware subyacente, manteniendo una considerable potencia para la programación a bajo nivel, pero, al añadir elementos que también le permiten un estilo de programación con alto nivel de abstracción, se consiguió un lenguaje híbrido y multiparadigma; características que se aprovechan en el desarrollo del software sobre el que versa el proyecto.

2.3 Entorno de Desarrollo Microsoft Visual C++

Una vez conocido el lenguaje de programación que se va a usar en este proyecto, lo siguiente que el lector se preguntará es ¿dónde programamos? Pues bien, en este caso, el trabajo ha sido implementado íntegramente en el Sistema Operativo Windows de Microsoft, por lo que se ha usado un entorno de desarrollo propio de esta compañía.

Microsoft Visual C++ [27], también conocido como Visual C++, es el entorno de desarrollo integrado (IDE) elegido para el presente proyecto, ya que permite crear aplicaciones utilizando diversos lenguajes de programación como: C, C++ y C++/CLI.

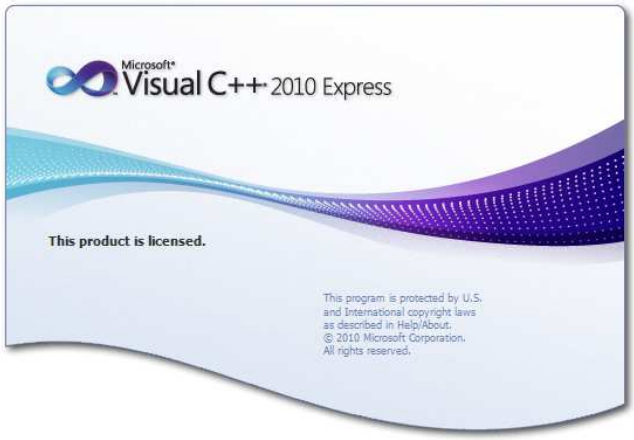

Figura 2.2: Microsoft Visual C++ 2010 Express Edition

Visual C++ es un IDE diseñado para el desarrollo y depuración de código escrito para las API's (*Application Programming Interface* o Interfaz de Programación de Aplicaciones) de Microsoft Windows, DirectX y Microsoft .NET Framework.

En concreto, para escribir nuestro software, la versión de Visual C++ utilizada es la Express Edition, la cual es gratuita y puede ser descargada desde el sitio web de Microsoft.

Una de las razones de su elección es que cuenta con herramientas como el IntelliSense, RemoteDebuging, Editar y Continuar, y Texto Resaltado, lo que facilita y agiliza enormemente la tarea de la programación.

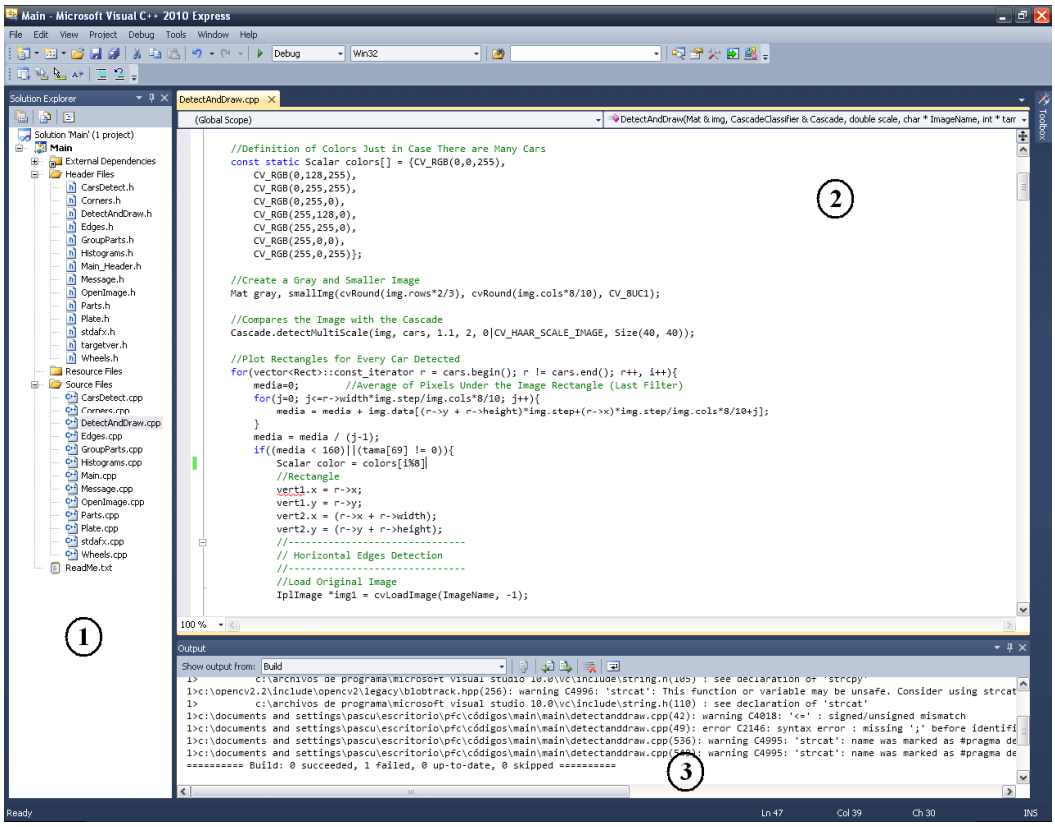

Figura 2.3: Ejemplo de programación en Visual C++

Como se puede ver en la figura 2.3, Visual C++ se estructura en tres zonas claramente diferenciadas:

1. El explorador de soluciones permite acceder a cualquier función o cabecera incluida en el proyecto actualmente en uso. Así como crear, eliminar o añadir nuevos archivos.

2. La ventana de programación es la zona principal del IDE, aquí es donde se escribe el código C++ de las distintas funciones.

3. La ventana de compilación muestra los resultados de la compilación e indica en que línea o líneas hay un error sintáctico. En la imagen, el error es debido a la omisión de un punto y coma que indica el final de una sentencia, por lo que se marca el error en la siguiente línea útil de código. También se puede observar que en la ventana de programación se ha resaltado automáticamente en rojo la zona donde se encuentra el error. Estos pequeños detalles hacen la programación mucho más rápida y eficiente.

2.4 Librerías OpenCV

Hasta ahora se ha hablado del entorno de desarrollo integrado donde ha sido programado este proyecto, así como el lenguaje de programación en el que está escrito, pero falta una última herramienta por comentar: las librerías OpenCV.

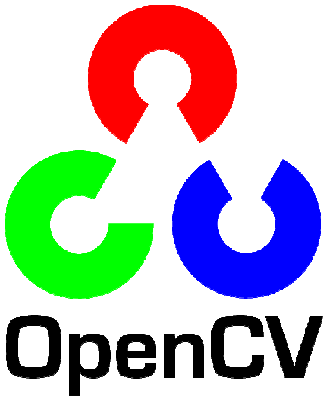

Figura 2.4: Logo OpenCV

OpenCV (*Open Source Computer Vision*) es una librería que contiene funciones de Visión por Computador en tiempo real [2] [15]. La primera versión alfa fue originalmente desarrollada por Intel en enero de 1999. En este proyecto se utiliza la biblioteca OpenCV 2.2.0 actualizada en 2011.

OpenCV es multiplataforma, es decir, las interfaces C, C++, Python y (próximamente) Java que contiene, funcionan a la perfección en diversos Sistemas Operativos: Windows, GNU/Linux, Android y Mac OS X.

Estas librerías contienen más de 500 funciones optimizadas, que abarcan distintas áreas en el proceso de Visión por Computador, como el reconocimiento de objetos, calibración de cámaras, visión estéreo y visión robótica avanzada. Gracias a esta diversidad, y a que OpenCV fue publicada bajo la licencia BSD, que permite su uso libre tanto para propósitos comerciales como de investigación, se ha utilizado en infinidad de aplicaciones, desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere el reconocimiento de objetos.

Las librerías OpenCV se dividen en cinco grandes grupos:

1. *CXCORE*: aquí se encuentran las estructuras y algoritmos básicos que usan las demás funciones, como la suma, media, operaciones binarias, etc.

2. *CV*: en este grupo están implementadas las funciones principales de procesamiento de imágenes, como el Operador Sobel comentado en el apartado 5.1.2

3. *HighGUI*: en él se encuentra todo lo relacionado con la interfaz gráfica de OpenCV, brinda herramientas para trabajar con imágenes y vídeos guardados en archivos u obtenidos directamente desde cámaras.

4. *ML*: aquí se encuentran algoritmos de aprendizaje y clasificadores, como el *adaboost* y el Clasificador de Haar, fundamentales en el proyecto y comentados en epígrafes posteriores.

5. *CvAux*: que contiene algoritmos experimentales

Todos estos grupos, a excepción del número cinco, han sido usados en algún momento durante el desarrollo del proyecto.

La justificación de por qué se han usado estas librerías de Visión por Computador en vez de otras como por ejemplo las librerías MIL de Matrox, es que, a parte de ser más completas y eficientes (en algunos casos), son fáciles de usar y permiten aprovechar las capacidades de los procesadores multinúcleo.

2.5 Clasificador de Haar

Dentro de este capítulo sobre la base teórica del proyecto, este apartado es uno de los más importantes, pues gracias a los Clasificadores de Haar se puede detectar casi cualquier objeto presente en una imagen [2].

Las librerías de OpenCV comentadas en el epígrafe anterior, implementan una versión del detector de rostros desarrollado por Paul Viola y Michael Jones [24] (conocido como *Viola & Jones detector*) y posteriormente ampliado por Rainer Lienhart y Jochen Maydt [11] para que use características diagonales.

En su trabajo, Viola & Jones demostraron como, a partir de características locales de la imagen basadas en el cambio de intensidad, se podía desarrollar un detector de rostros muy robusto. Partieron de la idea de determinar características usando sumas y restas de los niveles de intensidad de la imagen, para conseguir este objetivo usaron filtros de Haar (*Haar-like wavelets*). Funcionan aplicando filtros de diversos tamaños que recorren la imagen, si dicho valor está por encima de un cierto umbral, esa zona de la imagen se clasificará como cara.

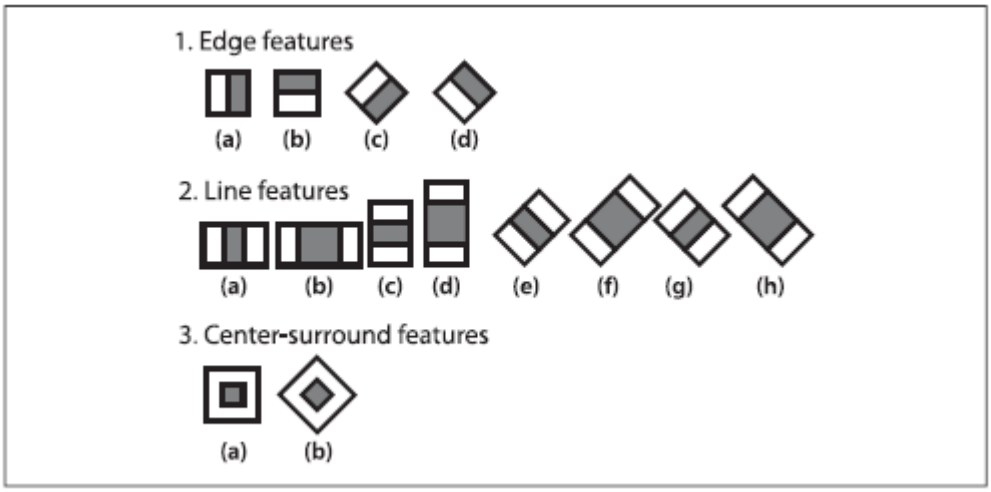

Figura 2.5: Filtros de Haar

Estos primeros clasificadores, por sí solos, consiguen resultados muy pobres pero, combinando varios de ellos se puede generar un clasificador mucho más robusto que incrementa exponencialmente el éxito de detección, esta técnica se conoce con el nombre de *boosting* (ver capítulo 3). De esta manera conseguimos clasificadores robustos muy precisos (del orden del 99.9% de acierto), el problema es que, además, presentan una elevada tasa de falsas detecciones (del orden del 50%).

Por este motivo, Viola & Jones propusieron un esquema basado en una cascada de clasificadores robustos.

En las primeras etapas de esta cascada de clasificadores se detectan zonas de la imagen que son muy diferentes de una cara, mientras que en las últimas etapas se rechazan ejemplos mucho más complejos.

Para cascadas de 20 etapas conseguimos una tasa de acierto de 0.999^{20} (a través de toda la cascada), es decir, del 98% aproximadamente, mientras que la tasa de falsos positivos decae hasta el valor de 0.5^{20}, es decir, aproximadamente 0.0001%.

OpenCV denomina Clasificador de Haar al detector de Viola & Jones, dado que usa los Filtros de Haar (figura 2.5). Una cascada de Clasificadores de Haar se denomina Cascada de Haar.

El objetivo de Viola & Jones era la detección de rostros, pero las Cascadas de Haar no sólo detectan caras, también funcionan bastante bien con objetos “rígidos”, es decir, que tienen vistas muy distintas. De eso nos aprovechamos en este proyecto, ya que las diferentes vistas de un coche son muy distintas entre sí.

Capítulo 3

Entrenamiento de una Cascada de Haar

Ahora que ya se sabe qué es una Cascada de Haar y para qué se utiliza en el proyecto, se pasará a profundizar más en estas ideas. Para ello, en este capítulo se explicarán los conceptos de *boosting* y *adaboost* en los que se basan las Cascadas de Haar. Posteriormente, en este mismo capítulo se comentarán los pasos necesarios para lograr la creación de una Cascada de Haar, esto es lo que se denomina Entrenamiento de una Cascada de Haar. Finalmente, en el último epígrafe se hace una introducción a la aplicación *Imageclipper* utilizada para agilizar el proceso de creación de estas cascadas.

3.1 Boosting

Es un método que consiste en combinar un conjunto de clasificadores sencillos o débiles que, por sí solos no conseguirían un porcentaje de detección aceptable, pero que combinados eficientemente producen un sistema de decisión muy robusto con tasas de detección elevadas y tasas de falsas alarmas de detección reducidas.

Los clasificadores débiles se denominan así por su sencillez y su escasa precisión. Se corresponden con reglas de clasificación simples que entregan como resultado un valor de confianza respecto a la predicción que están haciendo. Estos clasificadores evalúan características como los niveles de intensidad de una cierta zona de la imagen mediante sumas y restas para decidir finalmente si esa subimagen la clasifican como objeto o como fondo (no objeto).

Los clasificadores débiles generan resultados pobres, pero se pueden definir clasificadores más robustos enlazando varios de ellos, consiguiendo un sistema con una elevada robustez que podrá ser aprendido por una máquina mediante un proceso denominado entrenamiento. De este modo se habrá automatizado un sistema de decisión que permitirá encontrar los objetos buscados en el total de la información que contiene la imagen.

Cabe destacar que, aun cuando estos clasificadores robustos pueden llegar a una tasa de detección del 99%, la tasa de falsos positivos puede situarse por encima del 30%.

3.1.1 Adaboost

El *adaboost* fue presentado en 1995 por Yoav Freund y Robert Schapire [6] en la segunda Conferencia de Teoría de Aprendizaje Computacional y debe su nombre a la contracción de *Adaptative Boosting*, pues es un tipo de *boosting* que se ajusta adaptativamente a los resultados obtenidos por los clasificadores débiles de etapas previas, mejorando así su rendimiento global.

En otras palabras, es una mejora del *boosting* basada en su mismo principio, la creación de un clasificador fuerte mediante la combinación lineal de varios clasificadores débiles. La diferencia estriba en que el *adaboost* genera y llama a un clasificador débil nuevo en cada etapa, actualizando el valor de la distribución de pesos de estos clasificadores.

El *adaboost* funciona partiendo de un conjunto de imágenes de muestra a las que le aplica el método *boosting* de manera recursiva durante T etapas, quedándose en cada etapa con los mejores resultados y modificando el valor de los pesos de los clasificadores débiles que componen el *boosting* de tal forma que se le va incrementando el peso a aquellas muestras mal clasificadas y decrementando a aquellas bien clasificadas. Finalmente, al llegar a la etapa T, se genera un clasificador que combina los mejores clasificadores débiles de las etapas anteriores.

3.1.2 Gentle Adaboost

El algoritmo *adaboost* antes comentado proporcionó una nueva línea de investigación, llegándose a desarrollar distintas variantes del original *adaboost* como el *real adaboost* de Schapire & Singer en 1999 (una generalización del *adaboost*), o el *gentle adaboost* de Lienhart, Kuranov & Pisarevsky en 2002.

El *gentle adaboost*, como Lienhart et al reflejaron en su artículo [11], es el algoritmo que mejores resultados empíricos ante problemas reales ha dado, y es por eso por lo que ha tenido gran éxito. Es el que se utiliza en el presente proyecto, emplea el algoritmo que se detalla en la siguiente página para seleccionar el mejor CART (*Classification And Regression Tree*) simple que cumple con la tasa de detección escogida inicialmente.

Un CART es un árbol de clasificación y regresión, esto es, un modelo de predicción lógico que sirve para representar y categorizar una serie de condiciones que ocurren de forma sucesiva para la resolución de un problema. Toma como entrada un objeto o una situación descrita por medio de un conjunto de atributos y, a partir de esto, devuelve una respuesta relacionada con la entrada.

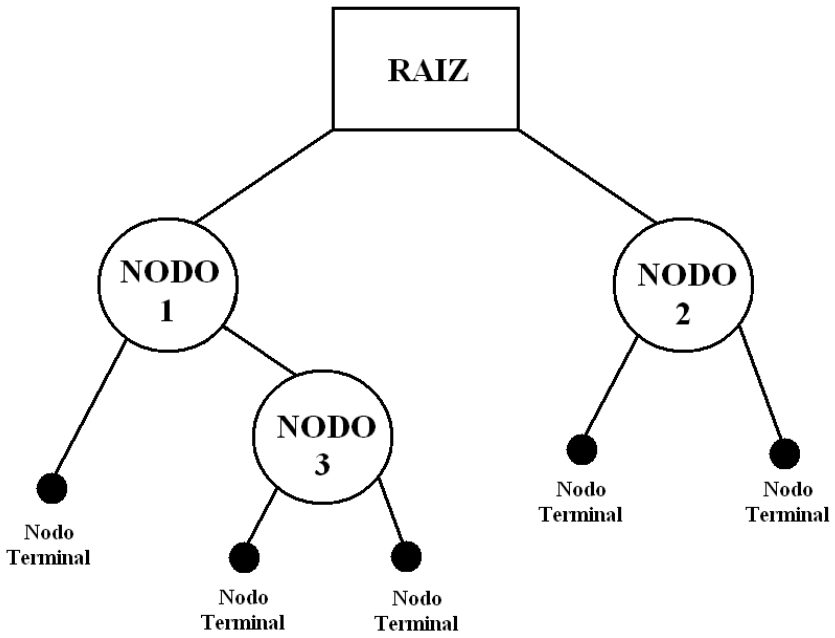

Figura 3.1: Ejemplo árbol de clasificación y regresión (CART)

El algoritmo de aprendizaje del *gentle adaboost* [14] está basado en N muestras $(x_1, y_1), \ldots, (x_N, y_N)$, donde x_i son las imágenes e $y_i \in \{-1,1\}$ las salidas de los distintos CART. Al comienzo de la fase de aprendizaje, los pesos ω_i son inicializados con el valor $\omega_i = 1/N$, luego, para seleccionar el mejor CART simple se repiten los tres siguientes pasos hasta que la tasa de detección seleccionada es alcanzada (desde t = 1 hasta t = T):

1. Cada clasificador simple es ajustado a los datos mediante mínimos cuadrados y se calcula el error respecto de los pesos ω_i.

2. Se elige el mejor CART h_t y se incrementa el contador t.

3. Se actualizan los pesos con el siguiente valor: $\omega_i = \omega_i \cdot e^{-y_i h_t(x_i)}$ y se vuelven a normalizar para que el sumatorio de los pesos sea igual a la unidad.

La salida final del clasificador es $h(x) = sign(\sum_{t=1}^{T} h_t(x))$, valor en el que se basa la construcción de la cascada de clasificadores.

3.2 Pasos del Entrenamiento

Una vez conocido el fundamento teórico y matemático de las Cascadas de Haar se puede proceder a explicar cómo se realiza el entrenamiento de estas cascadas [2], es decir, cómo se crean o, mejor dicho, cómo se consigue que un computador sea capaz de reconocer objetos haciendo uso de un único archivo y un programa que lo ejecuta.

Para que una Cascada de Haar funcione es necesario entrenarla mediante muestras. Se necesitan 2 tipos de muestras: positivas y negativas.

Las muestras positivas son imágenes recortadas que contienen el objeto a detectar. En nuestro caso, las muestras positivas deben ser un conjunto de imágenes recortadas con vistas traseras de diversos coches.

Las muestras negativas son imágenes que NO contienen el objeto a detectar, es decir, fondos. Lo más apropiado para este proyecto son imágenes de carreteras vacías o señales de tráfico, aunque, en general, cualquier imagen que no contenga el objeto de interés es válida.

Para un buen funcionamiento de la cascada es necesario recolectar entre 1.000 y 10.000 muestras de cada tipo.

· **Primer Paso: Creación de los Ficheros Index**

Una vez se tengan las muestras, lo siguiente es generar dos archivos *index* (.idx), para ello se puede usar el bloc de notas o cualquier otra aplicación de escritura informática.

Para los positivos, el archivo de texto debe ser de la forma:

<path>/img_name_1 count_1 x_{11} y_{11} w_{11} h_{11} x_{12} y_{12} . . .

<path>/img_name_2 count_2 x_{21} y_{21} w_{21} h_{21} x_{22} y_{22} . . .

. . .

Donde <*path*> es la ruta del directorio donde se encuentra la imagen e *img_name_1* es el nombre de la imagen (con su extensión). A continuación se debe indicar el número de objetos de interés presentes en la imagen (*count_1*) y su correspondiente ubicación dentro de la imagen mediante las coordenadas x-y del vértice superior izquierdo, el ancho y el alto de un rectángulo que lo contiene, es decir, x_{11} y_{11} w_{11} h_{11} respectivamente.

Por ejemplo, si la colección de imágenes está ubicada en el directorio *C:\imagenes\coches*, el fichero *index* tendrá un aspecto parecido al que sigue:

C:\imagenes\coches\coche1.jpg 1 65 99 25 37

C:\imagenes\coches\coche2.jpg 1 140 112 28 45

. . .

O, si se dispone de imágenes con más de un vehículo por foto:

C:\imagenes\coches\coche3.jpg 2 153 220 45 48 14 28 50 52

. . .

Una vez escrito el archivo de texto, sólo queda guardarlo como *index*; para ello se escribe el nombre que se desee seguido de la extensión .idx Por ejemplo, llamando *cars.idx* al archivo se tendría algo como la figura 3.2:

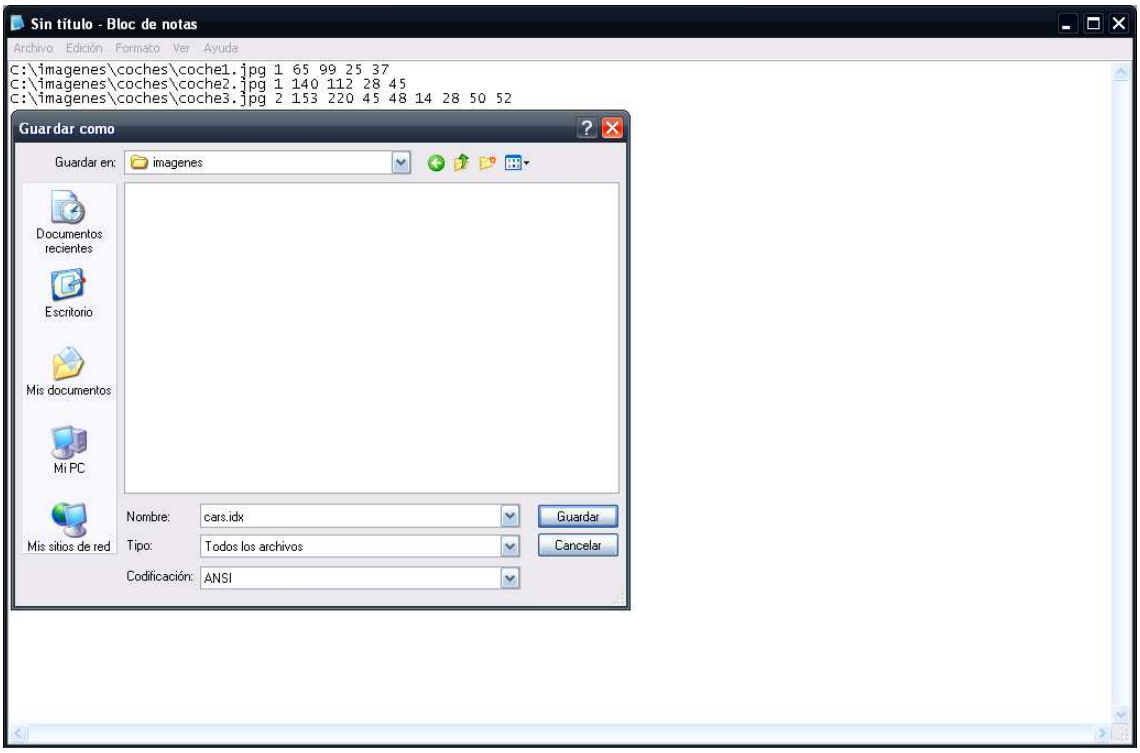

Figura 3.2: Ejemplo fichero index

Y al darle a guardar, se conseguiría el primer fichero *index* buscado.

Cabe destacar que existe una manera alternativa y más rápida de generar las muestras positivas y el archivo *index*; esto se explicará en el epígrafe 3.3: Imageclipper.

Para generar el segundo archivo se hace uso de los negativos. El proceso es similar al anteriormente explicado, salvo por el hecho de que en los negativos no está presente el objeto de interés y, por tanto, no se deben añadir los términos *count_1* x_{11} y_{11} w_{11} h_{11}, quedando el archivo de texto como sigue:

C:\imagenes\fondos\fondo1.jpg

C:\imagenes\fondos\fondo2.jpg

. . .

Se elige un nombre para el archivo y se guarda con la extensión .idx.

· **Segundo Paso: Creación de un Fichero Vector**

Por ahora nos olvidamos del archivo *index* generado mediante las muestras negativas, es decir, se usará el fichero *cars.idx* a continuación.

Con el fichero de las muestras positivas se debe generar un fichero vector (.vec) que se usará para entrenar la cascada. ¿Cómo se crea?, muy sencillo, usando la aplicación incluida en las librerías de OpenCV: *opencv_createsamples*.

La aplicación *opencv_createsamples* extrae las muestras positivas de las imágenes, las normaliza y las reescala al tamaño indicado y genera un archivo de salida con extensión .vec con el que poder entrenar la cascada.

Hay que decir que esta aplicación tiene más funcionalidades que no se han usado, como la posibilidad de aplicar transformaciones geométricas a las muestras, añadir ruido, alterar colores, etc. que permite incrementar el número de muestras "diferentes" de las que se dispone. Estas opciones son muy útiles en caso de no disponer de un relativamente elevado número de muestras positivas, o para detectar un objeto muy concreto, como pudiera ser un logo de una compañía, ya que se evalúan distintos ángulos, rotaciones y distorsiones en la imagen.

Volviendo al archivo vector, para generarlo, un ejemplo de la línea de comandos a escribir desde el terminal del sistema operativo sería:

```
C:\OpenCV2.2\bin> opencv_createsamples -vec cars.vec -info cars.idx -w 20 -h 20
```

Donde *cars.vec* es el archivo de salida que se obtendría y *-w 20 -h 20* es el tamaño (ancho y alto) del reescalado de cada muestra extraída.

Destacar que, por defecto, la aplicación *createsamples* genera mil muestras o menos, es decir, si se dispone de más de mil muestras, y no se indica nada en la línea de comandos, *createsamples* no creará un archivo vector con el número de muestras que se tenga, para ello, se debe añadir a la línea de comandos:

```
C:\OpenCV2.2\bin> opencv_createsamples -vec cars.vec -info cars.idx -w 20 -h 20 -num 2500
```

Con lo que se generaría, por ejemplo, 2500 muestras. También, si se intenta crear un archivo vector con menos de mil muestras o indicándole más muestras de las que se dispone, la aplicación nos avisará con un *parse error*, este mensaje es sólo informativo, esto es, el archivo se generará correctamente, pero no con mil muestras o con el número de muestras que se le haya indicado, respectivamente, sino con tantas muestras como se disponga.

· **Paso Final: Entrenamiento y Generación de la Cascada**

Una vez obtenido el archivo *cars.vec* del paso anterior, ya se puede entrenar y crear una Cascada de Haar. Para ello, se utilizará otra aplicación suministrada con las librerías de OpenCV: *opencv_haartraining*.

La aplicación *opencv_haartraining* genera una Cascada de Haar extrayendo muestras positivas según se indica en el archivo vector, y muestras negativas aleatorias indicadas en el archivo *index* de los fondos. Para conseguir esto se debe escribir la siguiente línea de comandos desde el terminal del sistema operativo:

```
C:\OpenCV2.2\bin> opencv_haartraining -vec cars.vec -data haarcascade -w 20 -h 20 -bg backgrounds.idx -nstages 20 -nsplits 1 -minhitrate 0.998 -maxfalsealarm 0.5
```

Siendo *haarcascade* un archivo .xml de salida con la cascada resultante, es decir, lo que verdaderamente se busca después de todo el entrenamiento.

Los parámetros *-w 20 -h 20* (ancho y alto) deben coincidir con los que se eligieron en el segundo paso para que no se produzca un error, e indican lo mismo que antes, el tamaño de la extracción de cada muestra.

Backgrounds.idx es el fichero *index* con los fondos, el generado en el primer paso, *-nstages 20* significa que nuestra cascada constará de 20 etapas, cada una formada por un clasificador robusto (ver apartado 2.5: Clasificador de Haar). Cuanto mayor sea el número de etapas que se elija, mayor será el tiempo de cómputo para la generación de la cascada.

Por último, los parámetros *-nsplits 1 -minhitrate 0.998 -maxfalsealarm 0.5* definen características del clasificador de cada etapa, *-nsplits 1* significa el número de divisiones o ramificaciones hasta alcanzar el valor deseado; un valor de 1 crea un nodo con dos posibles caminos de salida, estando uno de ellos conectado al siguiente nodo (ver figura 3.3), mientras que un valor de 4 genera un árbol de decisión (*Classification And Regression Tree* o CART) mucho más complejo. Por supuesto, se puede incrementar este parámetro, con lo que se obtendrían mejores resultados, aunque un mayor valor significa un mayor tiempo de cómputo para la generación de la cascada.

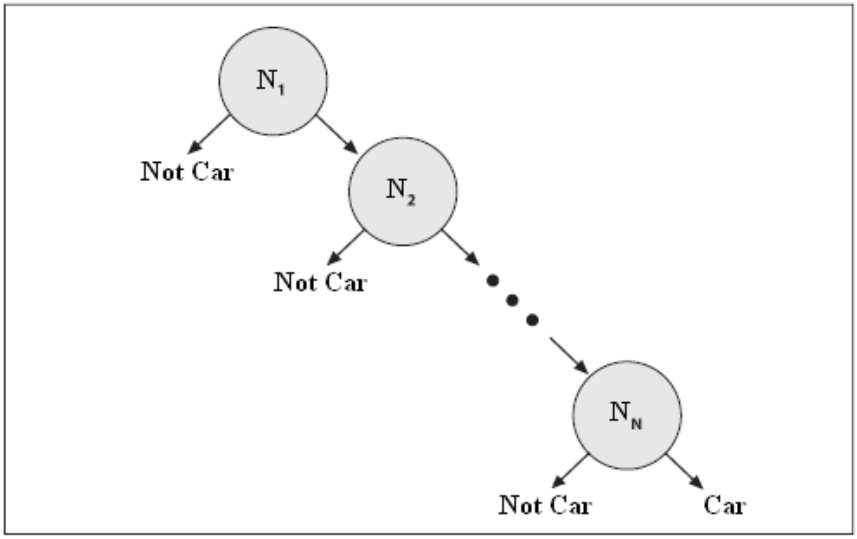

Figura 3.3: Cascada de rechazo con nsplits = 1

Los dos últimos parámetros indican, en tanto por uno, el mínimo valor de acierto y de fallo que deseamos, respectivamente. En el ejemplo se ha tomado una tasa de acierto del 99.8% y una tasa de fallo del 50% o inferior. Una vez más, un mayor índice de acierto o una menor tasa de fallo suponen un mayor tiempo de cómputo para la generación de la cascada.

Antes de ejecutar esta línea de comandos en el terminal, hay que tener muy presente que el entrenamiento de una cascada de Haar no es instantáneo, en párrafos anteriores se ha repetido varias veces que el valor de los parámetros influye en el tiempo de cómputo, pues bien, este es un hecho muy importante, porque en función de ellos y del número de muestras que se tomen, este proceso de entrenamiento puede llevar desde horas hasta semanas, incluso en los ordenadores más potentes.

Por supuesto, si el tiempo no es un factor decisivo, se puede crear una cascada especialmente precisa, pero, en general, hay que llegar a un acuerdo entre precisión deseada y tiempo, por lo que la elección de estos parámetros puede no ser tan trivial.

Cuando se haya decidido el valor de los parámetros ejecutamos la aplicación, para cada etapa se irá viendo algo similar a la figura 3.4.

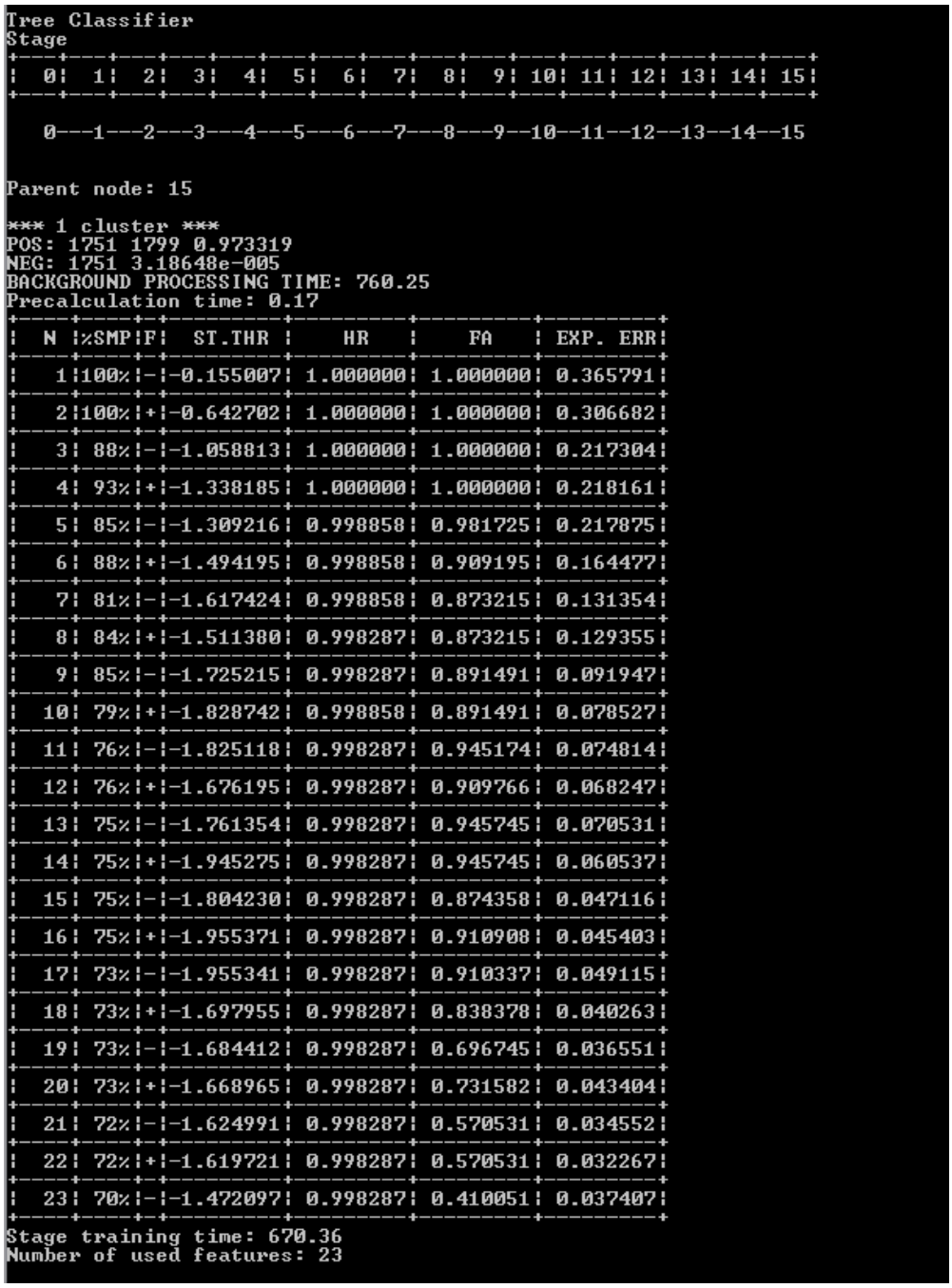

Figura 3.4: Ejemplo ejecución opencv_haartraining

En la figura anterior se muestra la etapa 15 de la creación de una cascada de 16 etapas con *nsplits = 1*, *minhitrate = 0.998* y *maxfalsealarm = 0.5*. Como se puede observar, la etapa termina cuando *maxfalsealarm* es menor que 0.5, ya que *minhitrate* va disminuyendo muy lentamente.

Cuando finalmente termina la ejecución de *opencv_haartraining* se mostrará un mensaje en el terminal como el de la figura 3.5.

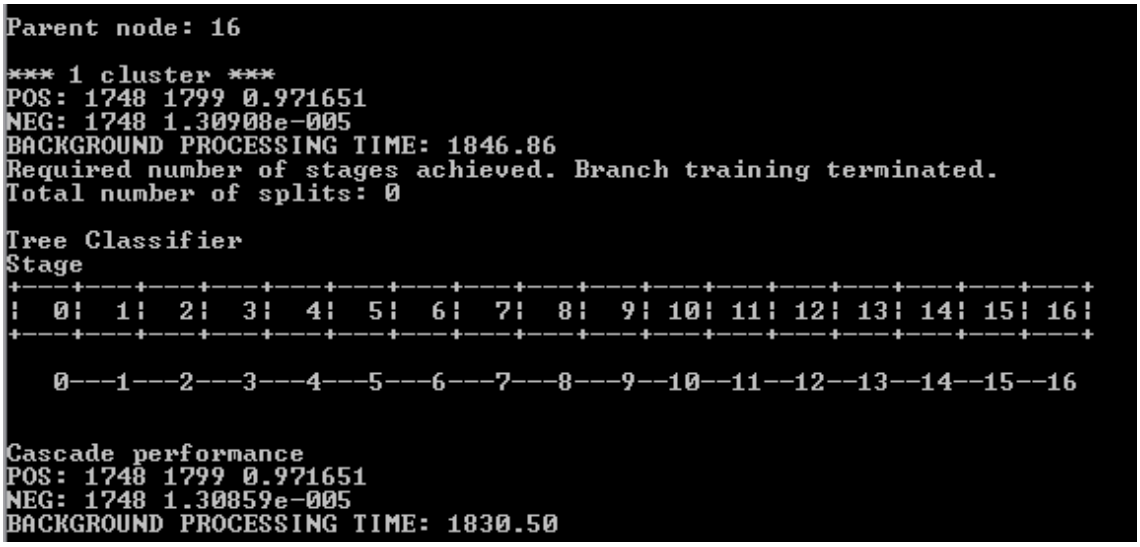

Figura 3.5: Ejemplo fin de ejecución opencv_haartraining

Después de salir del terminal se habrá creado un directorio con el nombre *haarcascade* y un archivo .xml también con el mismo nombre. Si se inspecciona este directorio se verá que está formado por varias carpetas numeradas desde cero hasta un número menos de etapas que se haya elegido para la cascada (ver figura 3.6).

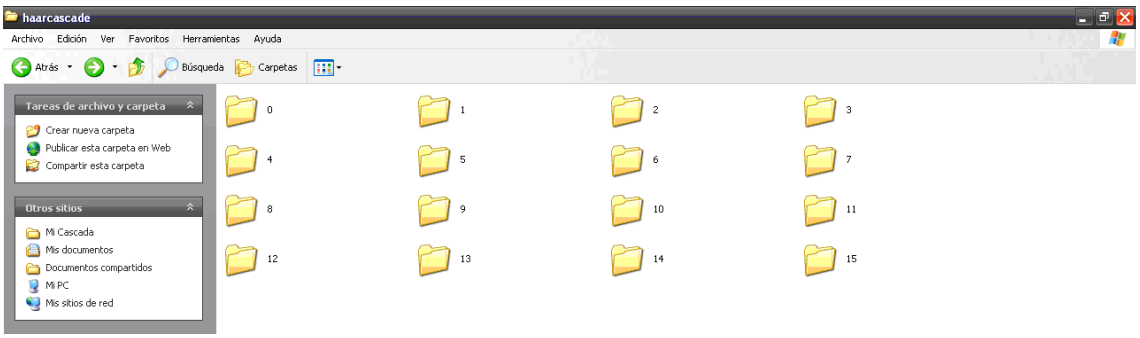

Figura 3.6: Ejemplo directorio de la cascada

Cada una de estas carpetas contiene un archivo de texto en el que se recoge la información del Clasificador de Haar robusto de esa etapa (ver figura 3.7), este clasificador se obtiene mediante la técnica del *boosting* (ver apartado 3.1), más concretamente la técnica del *adaboost*, desarrollada por Freund & Schapire en 1995 [6] y posteriormente generalizada por Schapire & Singer en 1997.

Siendo más concretos, en realidad, no se trata del *adaboost* puramente diseñado por Freund & Schapire, sino que se trata de una variante, el denominado *gentle adaboost* para CARTs comentado en el epígrafe 3.1.2. Esta variante, como se comentó en ese apartado, es la más exitosa en el procedimiento de detección de rostros según Lienhart y otros autores [11]; se basa en seleccionar un conjunto de CARTs simples para lograr alcanzar los valores de *minhitrate* y *maxfalsealarm* seleccionados.

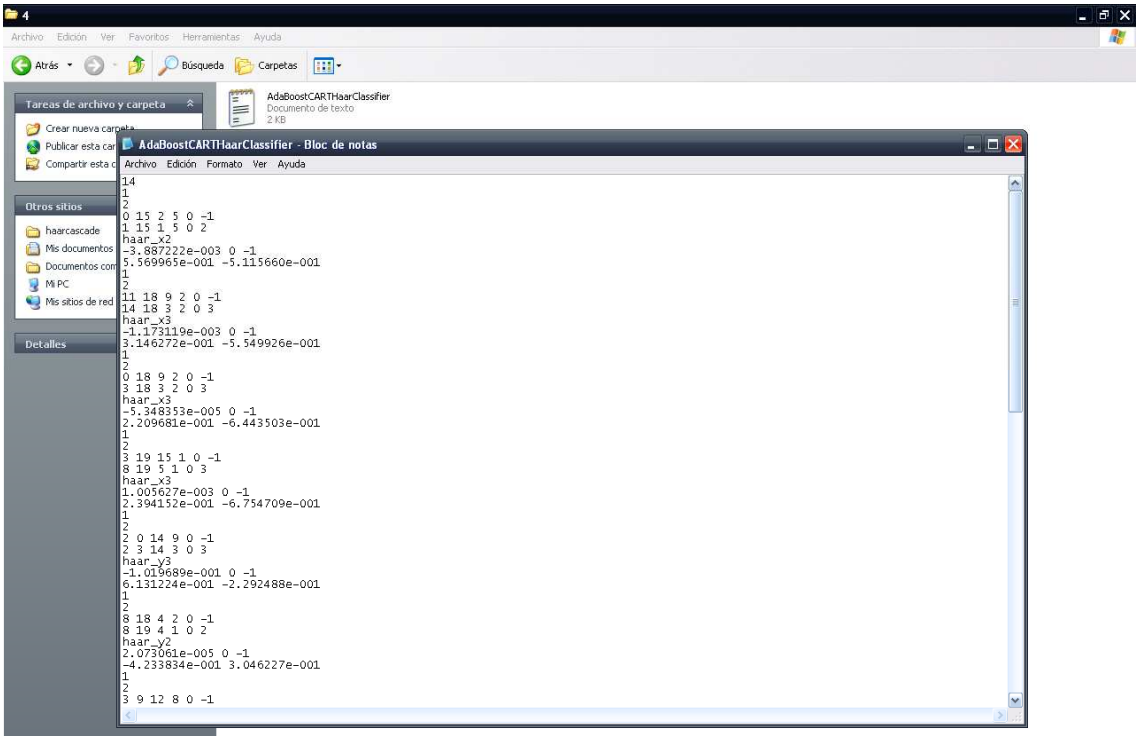

Figura 3.7: Ejemplo etapa de la cascada

Así pues, ya hemos conseguido crear nuestra propia Cascada de Haar para detectar vehículos. Las carpetas con los archivos de texto no serán de utilidad en el proyecto, sólo se utilizarán los archivos, como *haarcascade.xml* del ejemplo, resultantes del entrenamiento.

3.3 Aplicación Imageclipper

Gracias a las aportaciones al mundo científico de Naotoshi Seo [18] [19] se dispone de *Imageclipper*, una aplicación software libre que permite recolectar muestras positivas de forma manual pero muy rápida para el entrenamiento de nuevas Cascadas de Haar, es decir, posibilita extraer aquellas zonas de la imagen con el objeto de interés que se quiere detectar.

Este software permite:

- Abrir las imágenes de un directorio secuencialmente.
- Abrir un vídeo *frame* a *frame*.
- Seleccionar una zona de la imagen o del vídeo con el ratón y, posteriormente, guardarla y pasar a la siguiente imagen del directorio pulsando una única tecla: la barra espaciadora.

Estas características hacen de *Imageclipper* una herramienta de gran utilidad para nuestro proyecto.

También hay que destacar que *Imageclipper* guarda estas imágenes recortadas en formato .png, pero lo más importante y útil es la forma particular con la que renombra los recortes. Por ejemplo, si se tiene una imagen llamada *image.jpg*, el recorte podría tener un nombre como sigue:

```
image.jpg_0068_0066_0090_0080.png
```

Lo que permite generar el archivo de texto mencionado en el apartado 3.2, haciendo uso del siguiente comando:

```
$ find imageclipper/*_*_* -exec basename \{\} \; | perl -pe \
's/([^_]*).*_0*(\d+)_0*(\d+)_0*(\d+)_0*(\d+)\.[^.]*$/$1 $2 $3 $4 $5\n/g' \
| tee clipping.txt
```

Así pues, si se escribe el párrafo de la página anterior en la consola de comandos se obtendrá un archivo de texto con el nombre *clipping.txt* con la siguiente información:

```
image1.jpg 68 47 89 101
image2.jpg 87 66 90 80
image3.jpg 95 105 33 32
image4.jpg 109 93 65 90
image5.jpg 117 97 52 95
```

Una última opción explicada en [18] permite generar un archivo .idx o .dat (ambos son válidos para realizar el entrenamiento) sin necesidad de crear un archivo de texto intermedio, para ello, sólo se necesita tener las imágenes recortadas en un directorio, el archivo *haartrainingformat.pl* adjuntado con la aplicación *Imageclipper* y escribir en la consola de comandos la siguiente línea:

```
$ \ls imageclipper/*_*_* | perl haartrainingformat.pl --ls --trim --basename | tee info.dat
```

Lo cual generaría el archivo *info.dat* a partir del que se obtendría el archivo vector y la posterior cascada (ver apartado 3.2: Pasos del Entrenamiento).

Por último, decir que la aplicación, al ser libre, puede ser descargada gratuitamente desde [18].

Capítulo 4

Detección de Vehículos

Este capítulo y los dos siguientes forman parte del grueso del proyecto; en ellos se detalla cómo se usaron las herramientas comentadas en apartados anteriores para conseguir el propósito de este proyecto.

El presente capítulo está dedicado a la detección de vehículos; en él se explicará cómo se aplicaron Cascadas de Haar entrenadas mediante el proceso expuesto en la sección anterior. Destacar que, como se verá en las siguientes páginas, se han implementado dos métodos distintos para la detección de vehículos.

El primero, comentado a continuación, consiste en la detección de la parte trasera de un vehículo mediante una única Cascada de Haar; mientras que, el segundo, usará más de una Cascada de Haar para localizar distintas zonas de la parte trasera de un vehículo que, después, serán agrupadas mediante procedimientos de proximidad.

Por último, al final del capítulo se muestra cómo se dotó de memoria al programa para que recordase los vehículos detectados en imágenes previas, consiguiendo así un mejor funcionamiento de las cascadas, pues se evitan problemas como los falsos positivos.

4.1 Detección de la Parte Trasera de un Vehículo

El primer procedimiento que se ha utilizado para lograr detectar vehículos en secuencias de imágenes es el clásico método empleado para la detección de cualquier objeto rígido (según [2]) con vistas laterales y frontales muy distintas entre sí: la aplicación de una única Cascada de Haar para detectar un objeto concreto presente en una imagen digital.

En nuestro caso, la Cascada de Haar está entrenada para reconocer partes traseras de vehículos; para ello se recolectaron miles de fotos de gran diversidad de vehículos y se generó el archivo *index* correspondiente a las muestras positivas (ver apartado 3.2) haciendo uso de la aplicación *Imageclipper* de Naotoshi Seo explicada en el epígrafe 3.3.

También, siguiendo los pasos del entrenamiento de una Cascada de Haar del apartado 3.2, se recolectó miles de imágenes que no contuvieran vehículos, como fotos de carreteras vacías, señales de tráfico, marcas viales, etc. así como otra serie de fotos al azar, pero, como se ha dicho antes, sin ningún vehículo presente. Este conjunto de imágenes compone las denominadas muestras negativas o fondos. Posteriormente se almacenaron las muestras negativas en un directorio y se generó el archivo *index* correspondiente a los fondos.

Continuando con el entrenamiento, se generó el archivo vector de las muestras positivas y, finalmente, el archivo .xml con la cascada entrenada.

Una vez obtenida la cascada, el siguiente paso es implementar una aplicación escrita en lenguaje de programación C++ (ver apartado 2.2), para ello se usará el entorno de desarrollo integrado Microsoft Visual C++ comentado en la sección 2.3.

Antes de comenzar a programar, lo primero que hay que hacer es instalar las librerías OpenCV e incluirlas en el nuevo proyecto de Visual C++. Hecho esto, ya se puede comenzar a desarrollar la aplicación de detección de vehículos.

En la figura 4.1 de la siguiente página se muestra el diagrama de flujo del código del programa principal implementado en este proyecto.

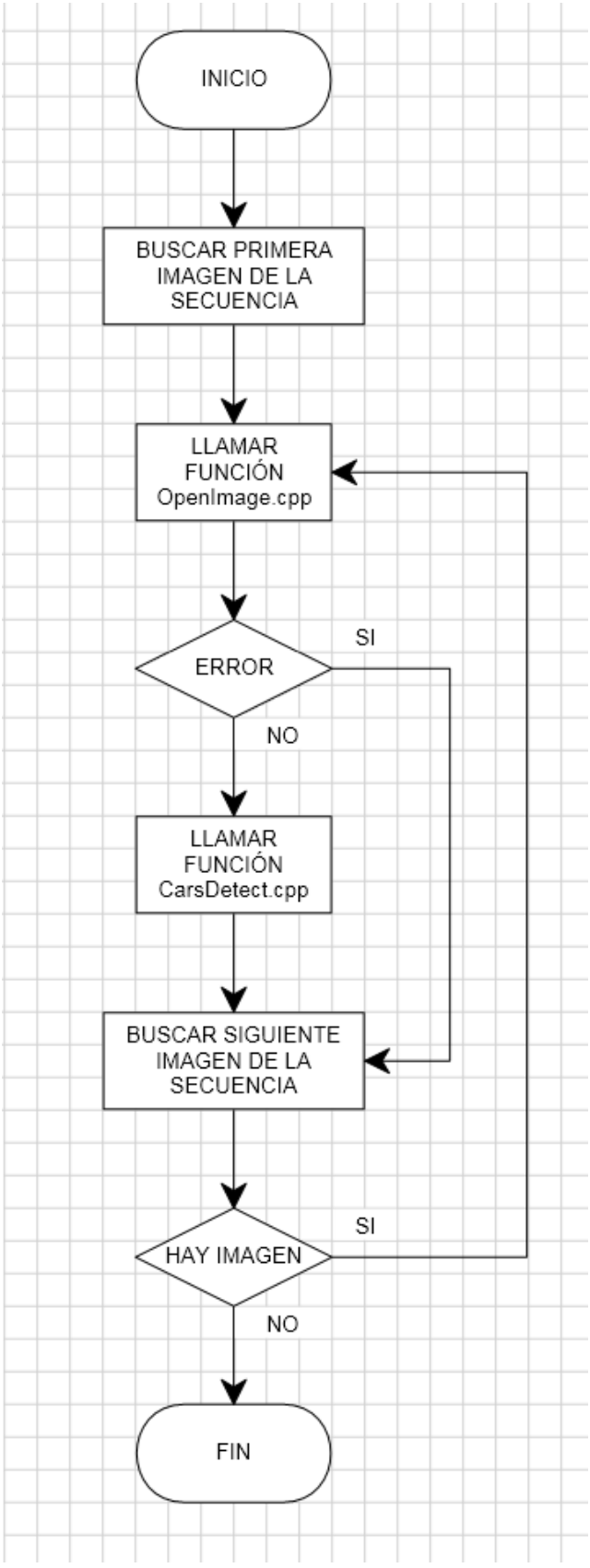

Figura 4.1: Diagrama de flujo de Main.cpp

El diagrama de flujo del programa principal o *main* realiza cinco acciones fundamentales: primero inicializa y define las variables a usar, después llama a la función *OpenImage.cpp* encargada de abrir las imágenes de un directorio, luego comprueba si ha habido algún error relacionado con la no existencia de una primera imagen para, posteriormente, llamar a la función *CarsDetect.cpp* en caso de que no haya habido ningún error. Finalmente, busca la siguiente imagen de la secuencia para realizar iterativamente estos pasos o, en caso de no haber más imágenes, finaliza la ejecución de la aplicación.

Antes de mostrar y explicar los diagramas de flujo del resto de funciones que componen esta primera parte de la aplicación se debe resaltar que, dado que este software va a ir implementado en el vehículo IVVI (ver capítulo 1), el que finalice la ejecución de la aplicación si no se encuentran más imágenes sería un error, es decir, la ejecución del software debe ser cíclica y sólo terminar en caso de pedirlo mediante una orden (o apagando el equipo). Esta consideración, así como otras que no se mencionarán en este capítulo, serán tratadas en el Capítulo 8: Aplicaciones del Proyecto, donde se explicará cómo amoldar a la realidad el trabajo realizado en el laboratorio. También hay que destacar que, para mayor detalle, los códigos del proyecto escritos en C++ pueden encontrarse en el apéndice de este documento.

Volviendo al tema que nos concierne, el diagrama de flujo de la función *OpenImage.cpp* se muestra a continuación:

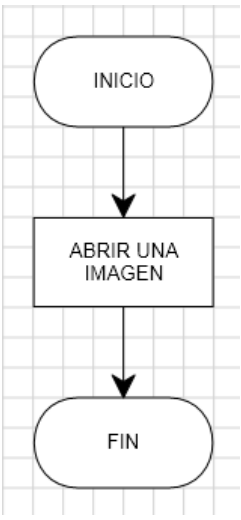

Figura 4.2: Diagrama de flujo de la función OpenImage.cpp

Como se ve en la figura 4.2, el diagrama de flujo de esta función es muy sencillo, *OpenImage.cpp* únicamente tiene como tarea abrir una imagen de la secuencia y comprobar que se ha podido cargar correctamente; en caso contrario mostrará un mensaje de error en el terminal.

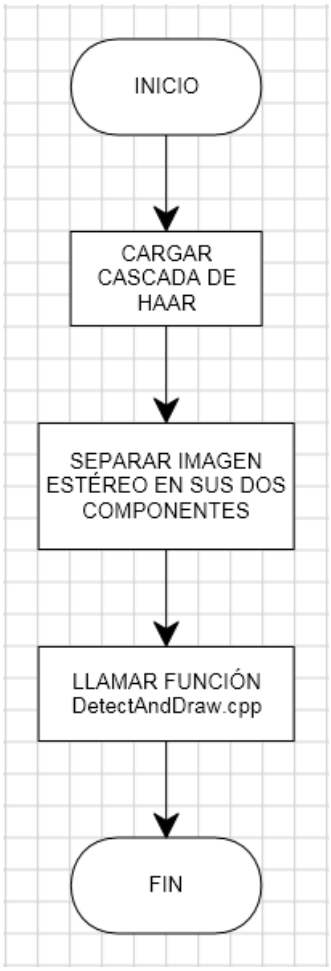

Figura 4.3: Diagrama de flujo de la función CarsDetect.cpp

La otra función llamada desde el programa principal es *CarsDetect.cpp* (ver figura 4.3), esta función es, por así decirlo, una plataforma de preparación para llamar a la función que verdaderamente realiza la detección de los vehículos presentes en la imagen: *DetectAndDraw.cpp*, pues primero carga la Cascada de Haar ubicada en el directorio "`C:/OpenCV2.2/Images/Cascades/`" (la ruta puede modificarse en caso de ser necesario) y después separa la imagen estéreo abierta por *OpenImage.cpp* en sus dos componentes sencillas: imagen tomada desde la cámara izquierda e imagen tomada desde la cámara derecha del sistema de visión estéreo.

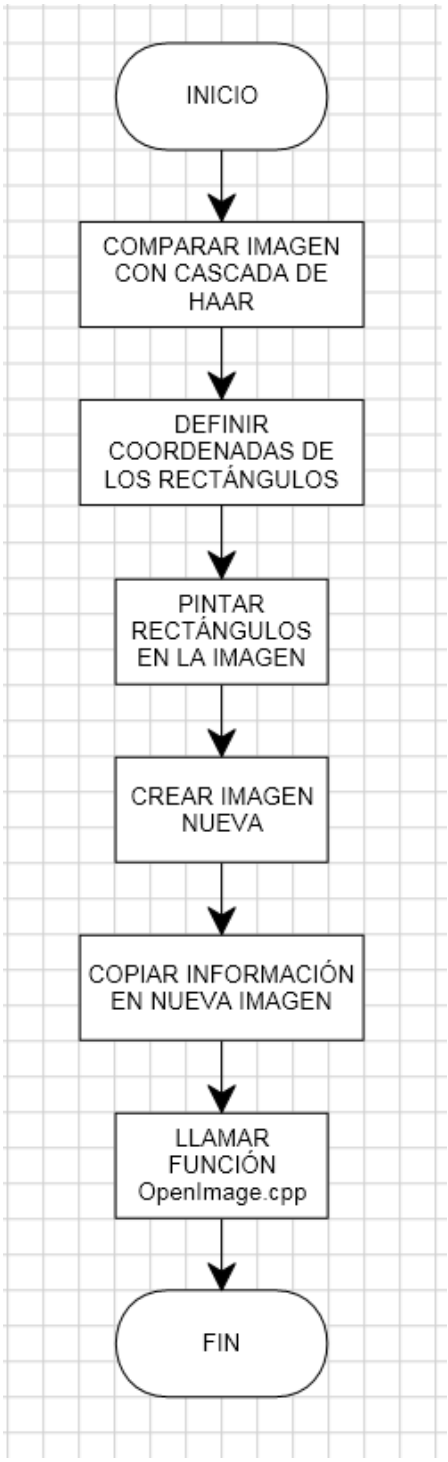

Figura 4.4: Diagrama de flujo de la función DetectAndDraw.cpp

Por último, se muestra en la figura 4.4 el diagrama de flujo de *DetectAndDraw.cpp*, esta función es el núcleo de la detección y, por consiguiente, de todo el proyecto. Este diagrama es el usado inicialmente; en mejoras posteriores será modificado y se corresponderá con el código C++ de la función *DetectAndDraw.cpp* del anexo.

Esta función realiza la comparación de la imagen actualmente procesada con la Cascada de Haar cargada por *CarsDetect.cpp*. Para ello se usa la siguiente función de las librerías OpenCV:

void CascadeClassifier::detectMultiScale(const Mat& *image*, vector<Rect>& *objects*, *1.1*, *3*, *0*, Size *minSize=Size()*)

Esta función detecta objetos de diferentes tamaños en la imagen y devuelve una lista de rectángulos donde se encuentran esos objetos. Sus parámetros no constantes se explican a continuación:

- *image*: matriz que contiene la imagen en la que se quiere detectar los objetos.
- *objects*: vector con los rectángulos que contienen los objetos detectados.
- *minSize*: indica el tamaño mínimo del objeto a detectar. Si se detectan más pequeños se ignoran.

Una vez hecha la llamada a esta función se obtiene una serie de rectángulos donde posiblemente hay objetos (hay que recordar que existe una tasa de acierto y otra de falsos positivos definidas durante el entrenamiento). Lo siguiente que se debe hacer es dibujar estos rectángulos en la imagen, para ello se usa otra función de la librería OpenCV:

void rectangle(Mat& *image*, Point *pt1*, Point *pt2*, const Scalar& *color*, int *thickness=1*, int *lineType=8*, int *shift=0*)

Que pinta rectángulos de un determinado color, grosor y tipo de línea en la imagen *image*, usando dos puntos: un vértice del rectángulo y su opuesto. Este proceso se debe hacer iterativamente para cada objeto detectado.

Ahora sólo queda copiar esta información en una nueva imagen para evitar corromper los datos de las imágenes capturadas por el sistema estéreo. Para ello se crea una nueva imagen igual a la original pero con otro nombre y se escribe esta información en ella.

Por último se realiza una nueva llamada a la función *OpenImage.cpp*, pero esta vez con la nueva imagen creada, para así poder ver los resultados de la detección.

4.1.1 Resultados

El programa implementado según los anteriores diagramas de flujo, es decir, sin memoria (ver sección 4.3) dio resultados como los que se muestran a continuación en este apartado.

Primero, en la figura 4.5 se muestra una imagen estéreo captada por el sistema de visión del IVVI para ilustrar cómo son este tipo de imágenes.

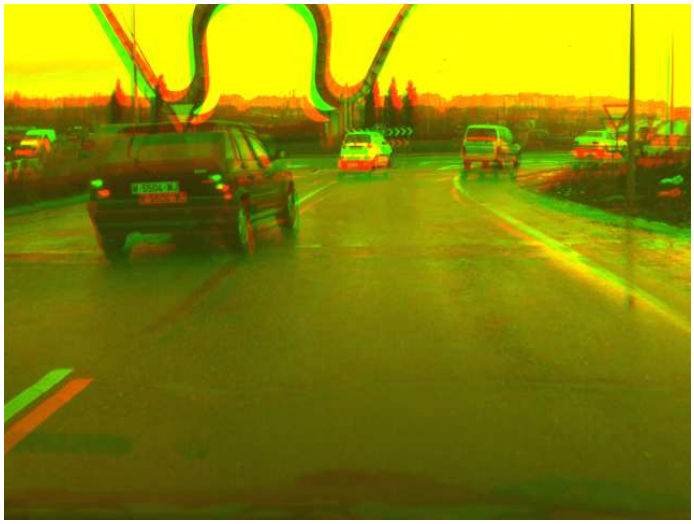

Figura 4.5: Imagen estéreo tomada por el IVVI

Ahora que nos hacemos una idea de cómo son estas imágenes, en la figura 4.6 se muestra la división de esta misma imagen estéreo en sus componentes: izquierda y derecha.

Figura 4.6: Componentes izquierda y derecha de una imagen estéreo

Como se observa, ambas imágenes son casi idénticas, la diferencia estriba en que han sido tomadas por dos cámaras separadas una cierta distancia horizontal, por lo que se percibe una sutil variación en el ángulo de perspectiva entre ellas.

Este proceso de separación lo realiza la función *CarsDetect.cpp* comentada anteriormente. Ahora, tomando como base una de estas dos imágenes (en el ejemplo siguiente hemos elegido la imagen derecha), se hace una llamada a *DetectAndDraw.cpp* consiguiendo resultados como los mostrados en la figura 4.7 y en la 4.8.

Figura 4.7: Resultado detección de vehículos con cascada 1

Figura 4.8: Resultado detección de vehículos con cascada 2

Estos resultados que se presentan en las figuras de la página anterior han sido obtenidos empleando dos Cascadas de Haar distintas. La primera tiene 15 etapas, mientras que la segunda ha sido entrenada con 20 etapas.

4.1.2 Comentarios

Es fácil darse cuenta de que, como se comentó en el apartado 3.2, a mayor número de etapas mejor funciona la Cascada de Haar, pues es mucho más precisa que otra entrenada con el mismo número de muestras pero con menor número de etapas.

Incluso, aunque se tenga el mismo número de muestras y estas muestras sean idénticas, si entrenamos una cascada con 15 etapas y otra con 20 etapas la cascada de mayor número de etapas, al estar formada por un mayor número de clasificadores robustos en cascada, tendrá menor cantidad de falsos positivos ya que éstos no pasarán la criba de las cinco últimas etapas y, por tanto, no se dibujarán en la imagen final.

También, a mayor número de etapas mayor tiempo de procesamiento de la imagen para obtener los rectángulos, por lo que hay que ser consciente de la capacidad de procesado de nuestro sistema de visión para no llevar un considerable retraso entre la captura y el procesado de nuevas imágenes; pues se formaría una cola de imágenes en espera de ser tratadas.

Por último destacar que no se debe elegir un número de etapas para el entrenamiento excesivamente alto, un valor adecuado ronda la veintena de etapas. Si se elige un número muy alto puede que la cascada sea tan estricta en el procesado de imágenes que haya objetos en la imagen y no los detecte por no considerarlos objeto. Es decir, se estaría particularizando demasiado el entrenamiento y la cascada sólo tendería a reconocer las muestras con las que ha sido entrenada.

A modo informativo hay que mencionar que, finalmente, después de muchos entrenamientos y pruebas de validación con diversos ejemplos, se optó por utilizar una Cascada de Haar desarrollada por otro alumno de la Universidad Carlos III de Madrid en su proyecto final de carrera, pues fue la que mejores resultados presentaba. Sus datos se citan en [7].

4.2 Detección de un Vehículo por Partes

Análogamente al procedimiento explicado en el apartado 4.1, se ha utilizado un segundo método para lograr detectar vehículos en secuencias de imágenes: la aplicación de varias Cascadas de Haar para reconocer un vehículo mediante sus partes. En otras palabras, se pretende encontrar un objeto como conjunto de sus partes.

Para ello, primeramente se entrenan las cascadas de las partes. En este proyecto se decidió detectar las ruedas (esquinas inferiores), las esquinas superiores y la matrícula del vehículo.

Estas cascadas no tienen nada distinto de lo ya explicado a lo largo del proyecto. Primero se realiza un entrenamiento a partir de una serie de muestras conteniendo las distintas partes de un vehículo, se generan los archivos *index* de las muestras positivas mediante la aplicación *Imageclipper* de Naotoshi Seo, y el archivo *index* de las muestras negativas (para las negativas se pueden usar las mismas muestras que para el apartado 4.1), se generan los archivos vector y, finalmente, se ejecuta *opencv_haartraining* para obtener tres archivos .xml, cada uno para su parte del vehículo correspondiente.

Una vez obtenidas las cascadas, al igual que se dijo en el apartado 4.1, el siguiente paso es implementar una aplicación escrita en lenguaje de programación C++ en el entorno de desarrollo integrado Microsoft Visual C++. Esta aplicación es muy similar a la que detecta zonas traseras de vehículos, la principal novedad es que se aplican tres cascadas sobre la misma imagen y se agrupan los resultados de estas cascadas para, finalmente, dibujar en la imagen aquellos rectángulos que cumplan una serie de condiciones impuestas.

Los diagramas de flujo de las funciones que componen esta nueva aplicación son idénticos a los mostrados en figuras anteriores, la única diferencia está en la función *DetectAndDraw.cpp* pero, aún así, se hace un breve recordatorio en las siguientes líneas.

El programa principal *Main.cpp* (ver figura 4.1) procesa la secuencia de imágenes y muestra la imagen analizada actualmente llamando a la función *OpenImage.cpp*, para después llamar a la función *CarsDetect.cpp* que, en este caso, sólo separa la imagen estéreo en sus dos componentes.

Una vez hecho esto, *CarsDetect.cpp* llamaría a *DetectAndDraw.cpp*, sin embargo, esta nueva aplicación se ha estructurado creando una función llamada *Parts.cpp*. Su diagrama de flujo puede verse a en la figura 4.9.

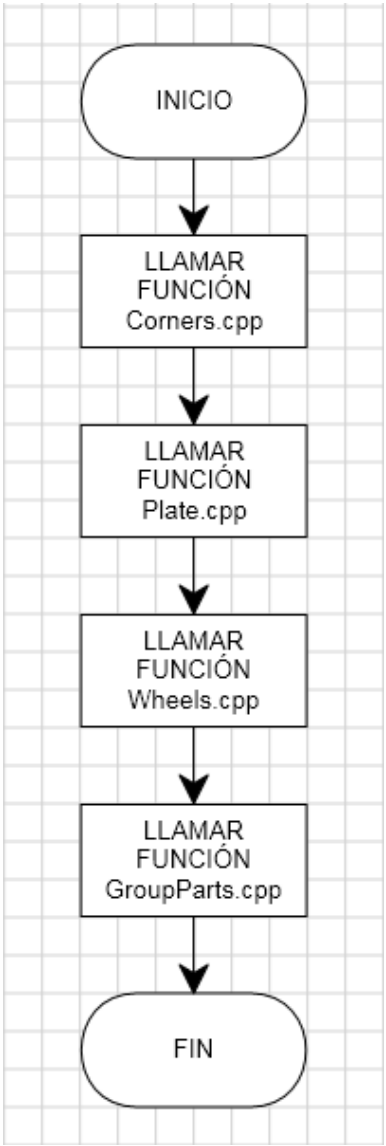

Figura 4.9: Diagrama de flujo de la función Parts.cpp

Esta función es en realidad una plataforma, pues desde ella se realizan llamadas a otras cuatro funciones: las tres primeras cargan y aplican a la imagen una Cascada de Haar correspondiente a una parte del vehículo, mientras que la última función se encarga de agrupar los resultados obtenidos por ellas y dibujar estos resultados sobre una copia de la imagen original.

4.2.1 Detección de Ruedas

En este subapartado y en los dos siguientes se comentarán brevemente las funciones que detectan cada parte independiente del vehículo. También se ha decidido ilustrar su funcionamiento mediante unas figuras con los resultados individuales de cada una de ellas para entender mejor de dónde sale el resultado final (ver figura 4.17) y cómo se efectúa la agrupación.

La detección de las ruedas se consigue gracias a la función *Wheels.cpp*. Esta función realiza las tareas mostradas en el diagrama de flujo de la figura 4.10.

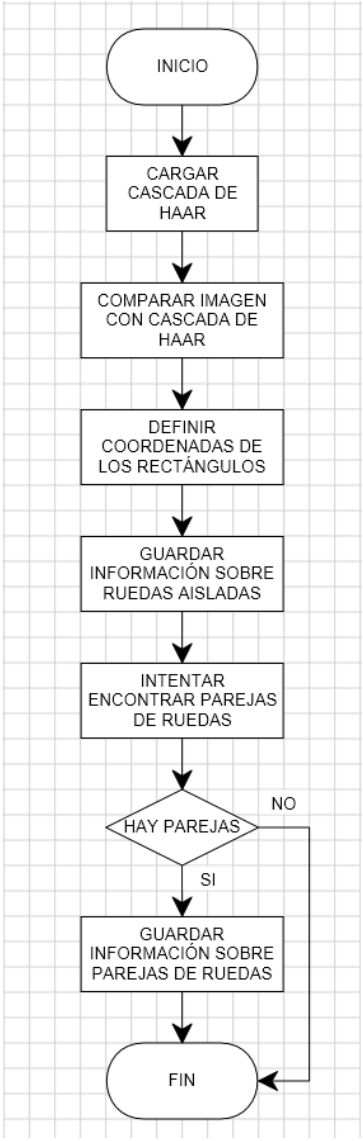

Figura 4.10: Diagrama de flujo de la función Wheels.cpp

Para la detección de las ruedas primero se carga la Cascada de Haar entrenada para detectar ruedas y luego se aplica esta cascada a la imagen actualmente abierta, consiguiendo así una lista de rectángulos con las posibles ruedas en la imagen.

Esta información se almacena en un vector que posteriormente se analiza en busca de una posible relación de proximidad entre ruedas; para ello, se define previamente un rango estandarizado para las posibles distancias entre ruedas. Este rango se consigue experimentalmente definiendo unos valores máximo y mínimo de distancias. Más concretamente, se usó como valores del intervalo: [35, 160] píxeles de distancia. Estos parámetros se establecieron extrayendo valores aleatorios del archivo *index* con las muestras positivas y creando un rango acorde a ellos. Recordemos que cada línea de los archivos *index* tenía una estructura como la siguiente:

```
C:\imagenes\coches\coche1.jpg 2 153 220 45 48 14 28 50 52
```

Donde el 2 indica que hay dos ruedas en la misma imagen (las dos del mismo coche) y los números 153 y 14 representan la coordenada X de la posición de dos rectángulos en la imagen conteniendo estas ruedas; por lo que haciendo una sencilla resta: *153 – 14 = 139*, se obtendría un valor estimado de la distancia.

Así pues, tras un periodo de estudio y extracción de muestras aleatorias se establece fácilmente el rango citado anteriormente.

En resumen, siguiendo el diagrama de flujo de la figura 4.10 y aplicando únicamente esta función se obtendrían resultados como el que se presenta en la figura 4.11.

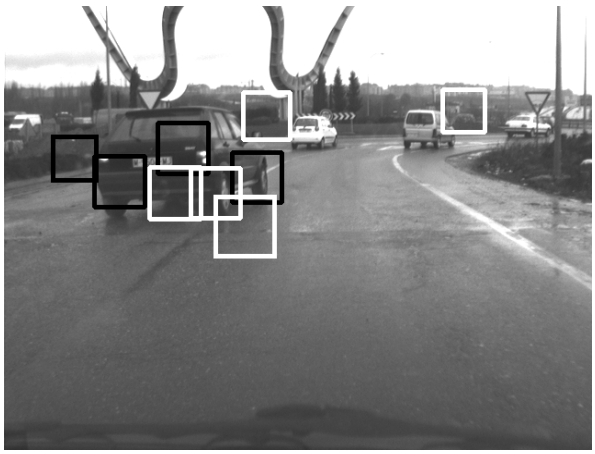

Figura 4.11: Resultado detección de ruedas

4.2.2 Detección de Esquinas Superiores

Análogo a lo comentado en el apartado anterior, se ha creado una Cascada de Haar para la detección de las esquinas superiores de los vehículos. Para utilizar eficazmente esta cascada se desarrolló otra función denominada *Corners.cpp*.

Corners.cpp sigue un diagrama de flujo similar al mostrado en la figura 4.10 para la función *Wheels.cpp*, por lo que se omitirá una figura con su diagrama de flujo. Esta similitud se debe a que ambas funciones realizan las mismas tareas pero con Cascadas de Haar distintas: comparan la imagen con la cascada, guardan los resultados e intentan encontrar pares de objetos, en el caso de *Corners.cpp*, pares de esquinas superiores.

Los pares de esquinas superiores, al igual que pasaba con los pares de ruedas, se agrupan mediante rangos, por lo que sería preciso definir otro para las esquinas; sin embargo, aunque generalmente los vehículos son más estrechos de la zona superior que de la zona inferior, se consideró la hipótesis de que los vehículos tienen su vista trasera cuadrada, por lo que se tomó el mismo rango de agrupamiento: [35, 160] píxeles.

La diferencia entre *Corners.cpp* y *Wheels.cpp* está en las variables donde almacenan los resultados pues, de ser iguales, se sobrescribiría la información.

Sin más dilación, a continuación se presenta una captura con el resultado de aplicar la función *Corners.cpp* exclusivamente.

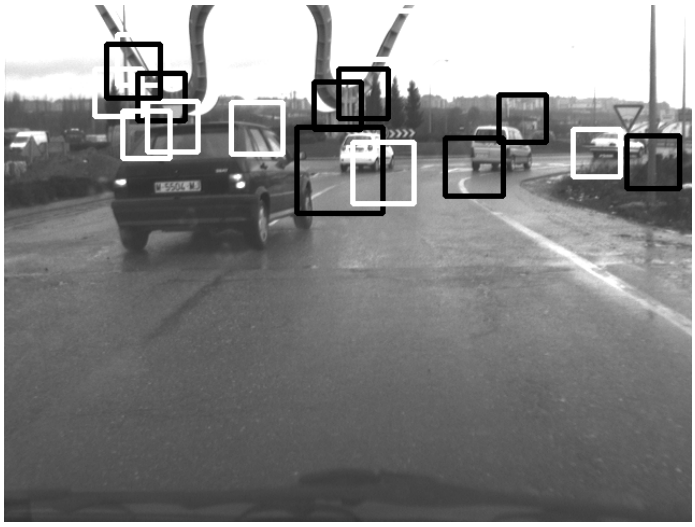

Figura 4.12: Resultado detección de esquinas superiores

4.2.3 Detección de Matrículas

Por último, se consideró crear una función que detectara las matrículas de los vehículos. Esta función, a parte de dotar de más robustez a la agrupación, podría ser modificada con posibles programas de reconocimiento de caracteres y, por tanto, de lectura de matrículas para búsqueda de vehículos robados, etc. por lo que se consideró de gran interés desarrollarla.

Retomando palabras antes escritas, la función *Plate.cpp* otorga más robustez a la agrupación, es decir, permite crear un sistema menos sensible ante falsos positivos y, además, permite valorar más casos en los que se considera que hay un vehículo presente en la imagen, como se explicará en el apartado 4.2.4: Agrupación de Partes.

La función *Plate.cpp* utiliza una Cascada de Haar que detecta matrículas, tal y como *Corners.cpp* y *Wheels.cpp* lo hacen, la diferencia entre la primera y las últimas se puede ver en el siguiente diagrama de flujo:

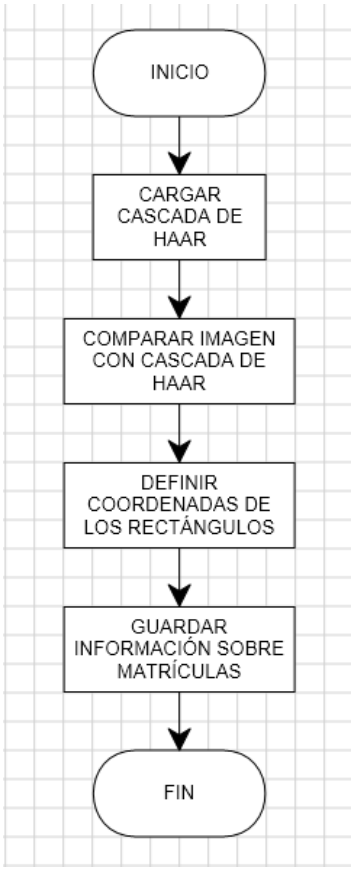

Figura 4.13: Diagrama de flujo de la función Plate.cpp

Por tanto, no se realiza ningún intento de agrupar pares de matrículas pues, como es bien sabido, los vehículos sólo disponen de una única matrícula en su parte trasera.

Así pues, ejecutando la función *Plate.cpp* en la misma imagen de la secuencia mostrada en anteriores figuras se obtendría:

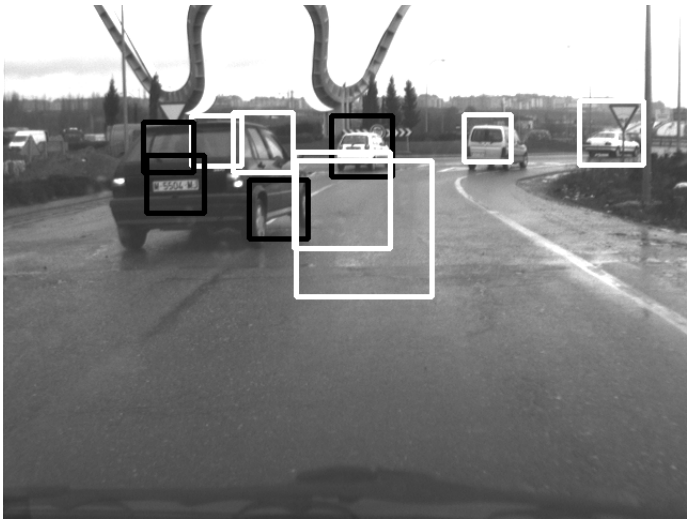

Figura 4.14: Resultado detección de matrículas

4.2.4 Agrupación de Partes

Una vez implementadas estas tres funciones de detección de partes, el siguiente paso es, como se muestra en la figura 4.9, llamar a la función *GroupParts.cpp* que se encargará de agrupar los resultados obtenidos previamente.

Esta función, si la comparamos con *DetectAndDraw.cpp*, es algo más compleja, pues no se puede limitar a pintar los resultados obtenidos por las cascadas sin más, también debe cribarlos para obtener mejores resultados finales pues, como se ha podido comprobar en figuras anteriores, las cascadas de detección de partes individuales tienen una elevada tasa de falsos positivos. Esto es debido al entrenamiento pero, principalmente, a la variedad y complejidad de formas, así como a su pequeño tamaño, lo que dificulta la búsqueda de las partes en la imagen.

Así pues, para solucionar el problema de las falsas detecciones se recurrió al diagrama de flujo presentado en la siguiente página.

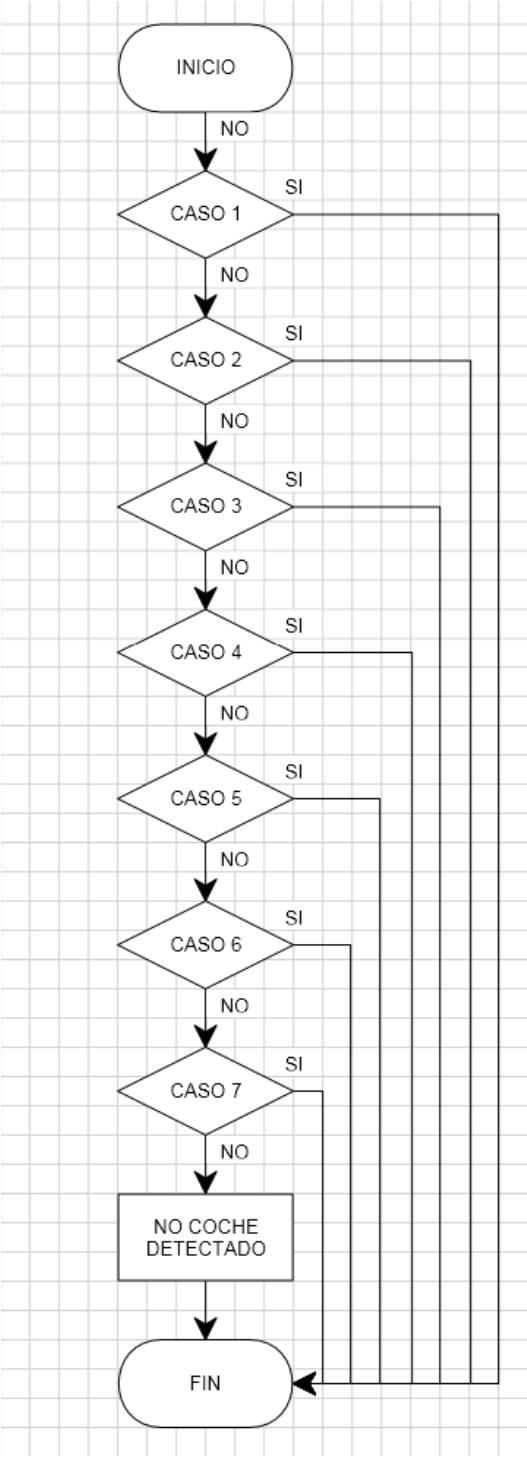

Figura 4.15: Diagrama de flujo de la función GroupParts.cpp

Como se observa, la función consiste en una serie de casos en cascada, cada uno con sus propiedades a cumplir (ver siguiente página). Si no se cumple ningún caso diremos que no se ha detectado vehículo alguno, es decir, las partes detectadas no se pueden relacionar entre sí, por lo que no habrá ningún objeto de interés en la imagen.

Si se cumple alguno de los casos que detallamos a continuación entonces se procede a analizar otro conjunto de rectángulos con partes de vehículos, pues puede haber más de un coche presente en la imagen. Como se verá, esta incertidumbre ante el número de vehículos presente en la imagen será uno de los factores clave, pues hará que no se reduzca tanto el número de falsos positivos como nos gustaría. En otras palabras, generalmente se agruparán más partes (y por tanto más vehículos) de los que en realidad hay en la imagen, pues se considerarán los casos de la figura 4.16, algunos, subconjuntos de los anteriores.

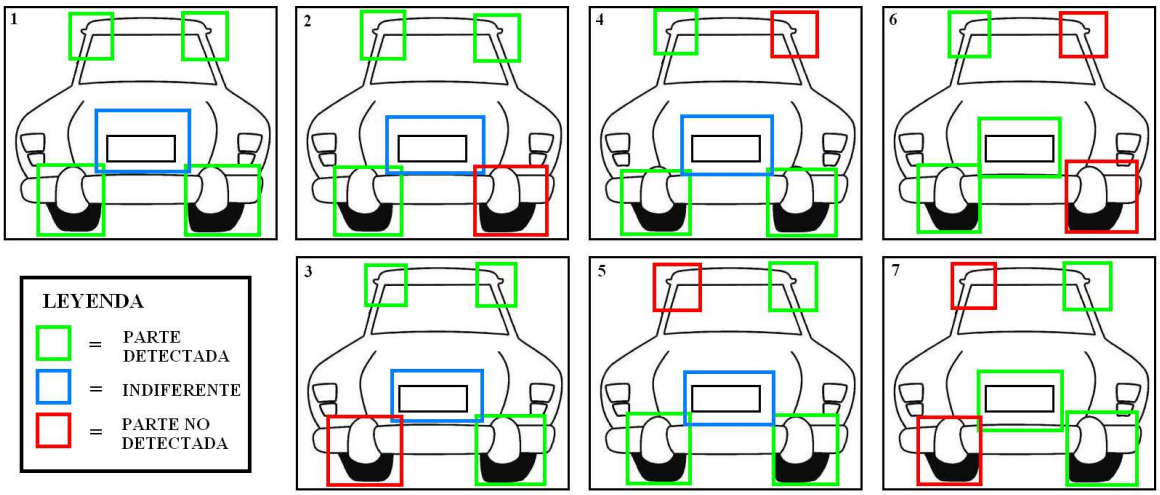

Figura 4.16: Casos de estudio para la función GroupParts.cpp

En la figura 4.16 se presentan los casos numerados en el diagrama de flujo desde el uno hasta el siete. Como se puede apreciar en la leyenda, los casos van desde haber emparejado pares de ruedas con pares de esquinas y con la matrícula (caso 1), hasta haber detectado únicamente la esquina derecha, la rueda derecha y la matrícula (caso 7).

Se han considerado solamente estos casos pues representan desde la detección perfecta de todas las partes del vehículo (caso 1), hasta un fallo en la detección o en la agrupación debido a la falta de una parte (casos 2, 3, 4 y 5), pasando por el caso particular de las oclusiones (casos 6 y 7) en el que un vehículo oculte parcialmente la zona trasera de otro vehículo, dejando entrever únicamente uno de los dos lados del coche.

Queda comentar el por qué de los rectángulos azules en la figura 4.16, pues bien, en los casos del 1 al 5 se ha considerado indiferente la detección de la matrícula porque no se ha querido restringir los resultados excesivamente, es decir, el que haya o no matrícula en esos casos no deja de significar que se ha detectado un vehículo. Eso si, si se detecta, añade robustez al sistema aumentando la probabilidad de que lo detectado sea un vehículo. Pero si no se detecta, puede haber sido porque la Cascada de Haar se entrenó mal o porque el vehículo no tiene la matrícula en la región donde se ha limitado la búsqueda, sea como fuere, la matrícula nos es indiferente en estos casos. Sin embargo, en los dos últimos casos, la matrícula se ha considerado fundamental para la clasificación como vehículo, ya que si sólo se detectaran una esquina y una rueda del mismo lado y esto se considerara vehículo, las falsas detecciones serían muy abundantes.

La agrupación de las partes se ha realizado mediante métodos de proximidad y rangos estandarizados (al igual que se explicó en el apartado 4.2.1). Para ello, y sabiendo en cada caso lo que se busca, se usa la información sobre los rectángulos almacenada en vectores tales como: un vector para las coordenadas y el tamaño de las esquinas aisladas detectadas, otro vector análogo al anterior pero con pares de esquinas, otro con ruedas aisladas, otro con pares de ruedas, y un último con las matrículas. Todos los vectores se procesan y comparan adecuadamente entre sí en los diversos casos para determinar cuales cumplen y cuales no. Para decidir si cumplen se han establecido unos rangos estandarizados que, al igual que para los pares de ruedas y esquinas, se obtuvo experimentalmente.

Para partes que van por parejas, como las esquinas o las ruedas, estos rangos ya se establecieron en sus respectivas funciones y se consideraron iguales según la hipótesis de que los vehículos tienen la vista trasera cuadrada. Se decidió por [35, 160] píxeles. Sin embargo, para agrupar esquinas con ruedas, se comprobó que los rectángulos de las cascadas podían estar aún más cerca que 35 píxeles, por lo que se modificó este rango a [20, 160] píxeles de distancia.

Finalmente, una vez sabido qué casos cumplen los rectángulos (suponiendo que cumplen alguno), el último paso es dibujar en la imagen los posibles vehículos detectados según cada caso.

4.2.5 Resultados

Una vez comprendido el funcionamiento de los programas que componen la función *Parts.cpp* se puede pasar a mostrar un resultado obtenido con esta aplicación de detección de vehículos como conjunto de sus partes.

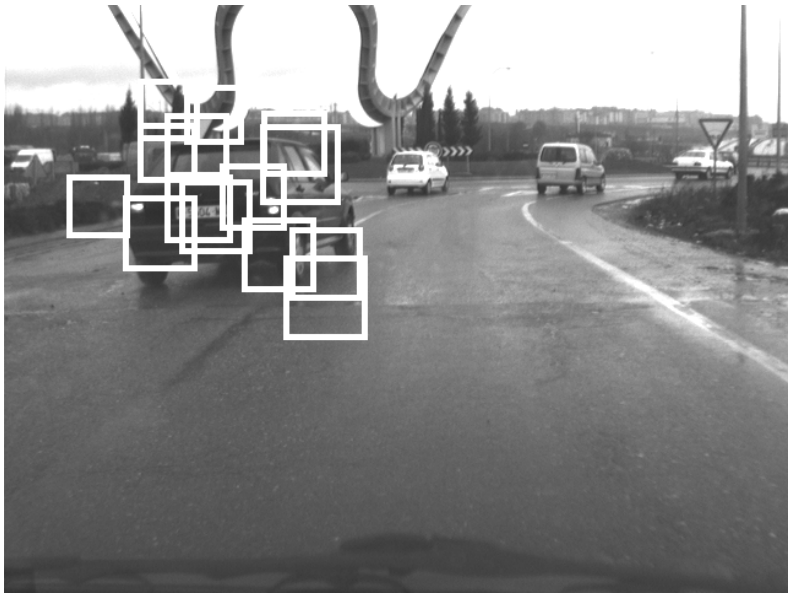

Figura 4.17: Resultado detección de vehículos por partes

Como se ve, el resultado final de la agrupación es un tanto confuso, pues se mezclan y dibujan todas las posibles combinaciones que cumplen alguno de los casos de la figura 4.16, sin embargo, si se observa detenidamente se pueden distinguir unos cinco rectángulos que se sitúan en la matrícula, las dos esquinas y las dos ruedas; por lo que el método, pese a no ser preciso, funciona correctamente.

También destacar que aquellos rectángulos aislados que aparecen en figuras previas son cribados gracias a este método, por lo que, como se ha dicho en el párrafo anterior, pese a que este método no es exacto, es bastante acertado, pues los rectángulos dibujados finalmente están todos en torno al vehículo en cuestión.

Por último, como apunte final decir que los otros vehículos más lejanos presentes en la imagen no son detectados, esto es debido a que el parámetro de la función *detectMultiScale* (ver página 53) del tamaño mínimo de detección lo hemos estimado en 40x40 píxeles.

No se ha implementado en este proyecto pero, para detectar vehículos lejanos se ha pensado que la mejor opción es aplicar la Cascada de Haar de la zona trasera del vehículo, pero reduciendo su campo y su tamaño de búsqueda para mejorar la aplicación en cuanto a tiempo de procesamiento.

4.2.6 Comentarios

Según se aprecia en la figura 4.17 y, comparando con la figura 4.8, a primera vista parece que el método de detección de un vehículo por partes no es tan preciso como el método de detección de la parte trasera completa; aunque esto es muy relativo pues depende de los entrenamientos de las Cascadas de Haar, pero lo que sí es cierto es que, para un mismo entrenamiento, es decir, para entrenamientos con el mismo número de muestras, de etapas y de tasas de detección y falsa alarma, la detección por partes funciona peor que la detección de la zona trasera del vehículo. Esto es debido a que los objetos buscados mediante las cascadas de las partes son de un tamaño más reducido que el vehículo entero y a que sus formas son más "comunes", en otras palabras, no son objetos tan distintivos como el trasero de un vehículo (excepto la matrícula, que, en general, ha dado siempre muy buenos resultados) y, por tanto, su detección presenta una mayor complejidad. No obstante, pese a ser un método menos preciso, es más robusto, pues el detectar un vehículo no depende de una única cascada, sino de la actuación de tres de ellas, así pues, si una falla o detecta peor, las otras dos restantes respaldan el resultado final.

Siguiendo con el tema de la poca precisión del método de detección por partes, hay que destacar que existen métodos para ayudar a aumentar la precisión, en el proyecto, como se explicará en el capítulo 6, se utilizan los mapas de disparidad para determinar distancias en la imagen y, además, para descartar parejas con distintos niveles de gris (y, por tanto, distinta profundidad en la imagen) para así reducir los falsos positivos en la imagen final y mejorar la precisión.

Por último comentar que también se desarrolló una Cascada de Haar para la detección de los faros traseros de los vehículos. El objetivo era dotar de mayor robustez a la detección por partes, pero se observó que los resultados individuales no eran tan buenos como se esperaba por la gran diversidad de formas existentes en los vehículos actuales, así pues, en vez de ayudar a mejorar la robustez empeoraba la precisión y los resultados globales, por lo que finalmente no se incluyó su uso en la detección por partes.

4.3 Detección con Memoria

En este apartado se explicará cómo se dotó de memoria a la detección para así lograr un mejor funcionamiento de la aplicación, pues, en base a la información previamente obtenida se puede perfeccionar la detección espacial en la imagen y lograr el entendimiento de su contexto y, por tanto, de la situación actual en la que se encuentra el vehículo dotado del sistema de visión estéreo: número de vehículos en la imagen, si se están aproximando o alejando estos vehículos, etc.

Primero, antes de comenzar con la explicación del algoritmo de memoria implementado, hay que destacar que la detección con memoria sólo se ha desarrollado para el método expuesto en el apartado 4.1: Detección de la Parte Trasera de un Vehículo, pues para la segunda aplicación resultaba bastante complejo e ineficaz el uso de la detección con memoria. Esto es debido al principal problema de la detección por partes: la escasa precisión. Como se ha comentado en apartados anteriores, a expensas de la precisión se conseguía robustez pero, para implementar la detección con memoria se necesita averiguar cuántos vehículos hay en la imagen, su tamaño y su posición X-Y aproximada, lo que resulta bastante complicado si se parte de una nube de rectángulos. La cantidad de restricciones y posibles casos que había que valorar hizo que finalmente se descartara el dotar de memoria a la detección por partes.

Así pues, centrándonos en la detección de vehículos mediante una única Cascada de Haar entrenada para reconocer partes traseras de vehículos, se desarrolló en código C++ el algoritmo de la siguiente página.

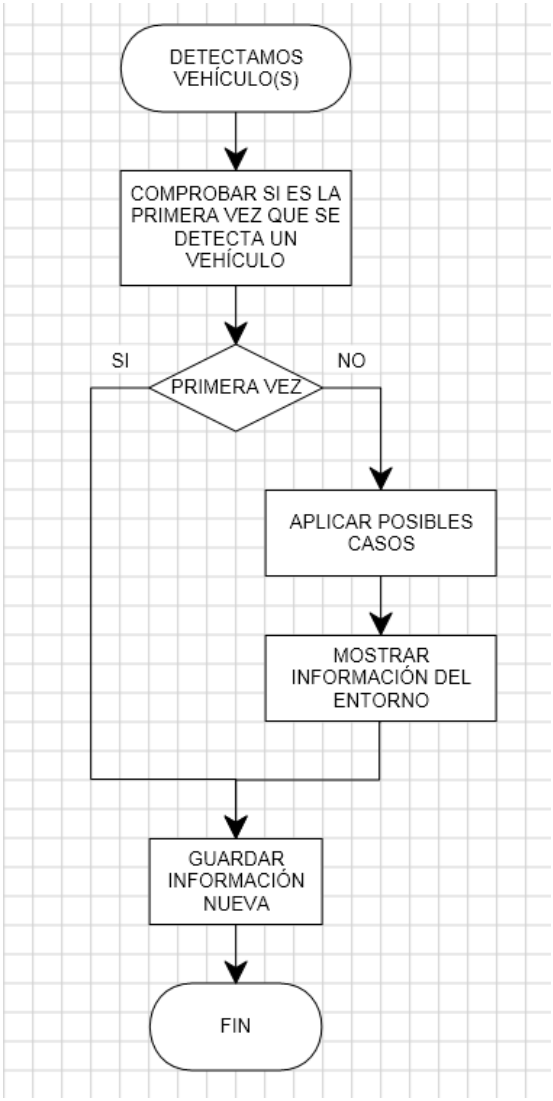

Figura 4.18: Algoritmo para la detección con memoria

El algoritmo de la figura 4.18 es bastante sencillo de comprender: primero se comprueba si es la primera vez que se detectan vehículos. Si es así se almacena información sobre su tamaño, su posición X-Y en la imagen y el número de posibles vehículos detectados, en caso contrario, quiere decir que ya se dispone de información previa (la almacenada cuando pasamos por la rama "SI" de la figura 4.18).

Así pues, cuando se llama a la función *DetectAndDraw.cpp* con la siguiente imagen de la secuencia se utiliza la información de la imagen anterior para, mediante los posibles casos de estudio que se pueden dar en la realidad (detallados a continuación), presentar por pantalla una descripción del entorno del vehículo: número de coches, si se acercan, permanecen a la misma distancia o se alejan.

Finalmente, la información vieja se borra y se sobrescribe con la información recién almacenada para utilizarse en la siguiente imagen de la secuencia, consiguiendo así una reducción de falsos positivos y un aumento de la precisión que, junto con el proceso de seguimiento explicado en el capítulo 5, dotarán a la detección de vehículos mediante una única Cascada de Haar de la robustez que carecía.

Para valorar los posibles casos que pueden ocurrir en la realidad se usa la información de la imagen previa, más concretamente el número de vehículos (obtenido mediante el recuento de cascadas dibujadas), el tamaño de estos vehículos (logrado mediante el ancho y alto de las cascadas) y la posición de los vehículos en la imagen (conseguida mediante las coordenadas X-Y de las cascadas). Esta información se compara con la nueva para comprobar qué casos se cumplen y cuáles no.

Como parámetro más importante se eligió el tamaño, pues comparando el área actual con las áreas almacenadas previamente se puede estimar qué coche se está analizando en cada momento o si se trata de uno nuevo (o una falsa detección). Para utilizar la información proveniente de los tamaños se compara el área actualmente en análisis con las almacenadas de la imagen anterior; si este valor coincide con uno (o varios) de los anteriores se incrementa un contador denominado *eq* (del inglés *equal*) en el código C++ (ver *DetectAndDraw.cpp* en el apéndice). Si este valor de área es superior a alguno de los anteriores se incrementa el contador *big* y si, por el contrario, es inferior, se incrementa el contador *sma* (del inglés *small*). Ahora, con los valores de los contadores se procede a estudiar los posibles casos mostrados a continuación:

1. Se cumple que $eq \geq 1$, son casos algo peculiares e improbables.

    1.1. Si el área y las coordenadas X-Y coinciden exactamente con alguna cascada previa quiere decir que nada ha cambiado. Ambas imágenes son idénticas, por lo que el coche y su entorno estará estático.

    1.2. Si el área coincide exactamente y las coordenadas X-Y son muy parecidas pero no iguales, entonces se trata del mismo vehículo detectado previamente pero desplazado en la imagen, por lo que puede ser un vehículo muy lejano o uno que se mueve a velocidad constante con nuestro vehículo.

1.3. Si no se dan los dos puntos anteriores se trata de una falsa detección.

2. Se cumple que $eq = 0$, casos más comunes.

2.1. Si se cumple que $sma = 0$

2.1.1. Si el área analizada y las coordenadas se parecen a alguna previa pero sin ser iguales, entonces nos encontramos con el vehículo más grande de la imagen. El vehículo más grande detectado se está acercando.

2.1.2. Si no se cumple el punto anterior, se trata de un nuevo vehículo o de un falso positivo, por lo que hay que guardar su información y procesarla en la siguiente imagen.

2.2. Si se cumple que $big = 0$

2.2.1. Si el área analizada y las coordenadas se parecen a alguna previa pero sin ser iguales, entonces nos encontramos con el vehículo más pequeño de la imagen. El vehículo más pequeño detectado se está alejando.

2.2.2. Si no se cumple el punto anterior, se trata de un nuevo vehículo o de un falso positivo, por lo que hay que guardar su información y procesarla en la siguiente imagen.

2.3. Si se cumple que $big > sma$ y que $sma \neq 0$

2.3.1. Si el área es ligeramente superior a alguna de las almacenadas y las coordenadas son parecidas, entonces un vehículo de tamaño intermedio se está acercando.

2.3.2. Si no se cumple el punto anterior se trata de una falsa detección.

2.3.3. Si el área es ligeramente inferior a alguna de las almacenadas y las coordenadas son parecidas, entonces el vehículo más grande de la imagen se está alejando.

2.3.4. Si no se cumple el punto anterior se trata de una falsa detección.

2.4. Si se cumple que $big < sma$ y que $big \neq 0$

2.4.1. Si el área es ligeramente superior a alguna de las almacenadas y las coordenadas son parecidas, entonces el vehículo más pequeño de la imagen se está acercando.

2.4.2. Si no se cumple el punto anterior se trata de una falsa detección.

2.4.3. Si el área es ligeramente inferior a alguna de las almacenadas y las coordenadas son parecidas, entonces un vehículo de tamaño intermedio se está alejando.

2.4.4. Si no se cumple el punto anterior se trata de una falsa detección.

2.5. Si se cumple que $big = sma$

2.5.1. Si el área es ligeramente inferior a alguna de las almacenadas y las coordenadas son parecidas, entonces un vehículo de tamaño intermedio se está alejando.

2.5.2. Si no se cumple el punto anterior se trata de una falsa detección.

2.5.3. Si el área es ligeramente superior a alguna de las almacenadas y las coordenadas son parecidas, entonces un vehículo de tamaño intermedio se está acercando.

2.5.4. Si no se cumple el punto anterior se trata de una falsa detección.

Por último, mencionar que también se tratan otros casos particulares como los ocultamientos o apariciones de vehículos. Para ello se incluyen nuevos bucles de decisión en los casos 2.3, 2.4 y 2.5 con la condición de que "si no se cumplen los puntos 2.3.1 y 2.3.3, o los 2.4.1 y 2.4.3, o los 2.5.1 y 2.5.3 entonces hay un ocultamiento o un falso positivo". Se activa pues una variable indicadora (*err = 2* en la función *DetectAndDraw.cpp*) que conduce a un bucle de decisión para finalmente elegir si se trata de un ocultamiento producido por otro vehículo o de un falso positivo.

Esta última decisión se toma en base a la localización espacial del rectángulo en la imagen, si se cumple lo mostrado en la figura 4.19 entonces se considera ocultamiento.

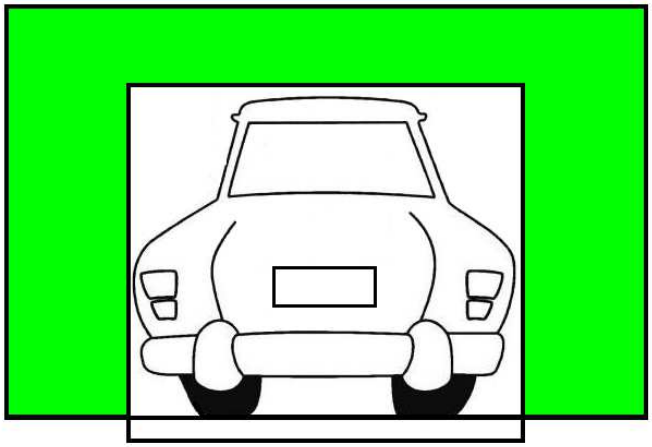

Figura 4.19: Caso especial de ocultamiento

Es decir, si las coordenadas del rectángulo analizado se encuentran en la zona verde entonces el vehículo estaba oculto por otro en la imagen anterior.

Capítulo 5

Seguimiento de Vehículos

En este capítulo se explicará otra parte fundamental del proyecto, el seguimiento de vehículos. Por seguimiento se entiende el acto de buscar un objeto en una imagen, para ello, previamente se tiene que haber detectado correctamente el objeto a buscar en las siguientes imágenes, es por esto que la exactitud en la detección de vehículos comentada en el capítulo 4 es tan importante.

Así pues, en esta sección se expondrá cómo se desarrolló un método para buscar y encontrar los vehículos detectados en las imágenes de la secuencia. Este método consiste en una serie de pasos consecutivos que se irán explicando en los diversos apartados del capítulo. Pero, para que el lector se haga una idea general sobre su funcionamiento, el método parte de la obtención de los contornos superior e inferior de cada vehículo; luego se traza una recta vertical centrada en el trasero del coche y que va de borde superior a borde inferior. Tomando esta recta como referencia se extrae un pequeño rectángulo ligeramente más ancho que la citada recta y se transforma convirtiéndolo en valores numéricos. Estos valores se procesan y, utilizando el rango intercuartílico (explicado más adelante) se obtiene un único valor. Este número será la huella dactilar del vehículo por así decirlo, pues en imágenes posteriores se buscará este valor (o el más próximo) para recolocar la recta vertical y, por tanto, la cascada de detección. En caso de no ser el mismo valor se almacenará éste último para hacer la búsqueda en la siguiente imagen. Este procedimiento se irá explicando poco a poco a lo largo del presente capítulo.

Hay que comentar que sólo se ha implementado un método de seguimiento pese a haber desarrollado dos procedimientos de detección de vehículos, la razón es que este método, salvo pequeñas modificaciones en el código, es ampliamente intercambiable con ambas técnicas. Para comprender el funcionamiento del seguimiento se explicarán todos los pasos con el procedimiento de detección de vehículos mediante una única Cascada de Haar (ver apartado 4.1), ya que este método es más sencillo y permitirá una mejor visualización y entendimiento del seguimiento.

En los últimos apartados se muestran imágenes con los resultados del seguimiento de un vehículo, así como unos breves comentarios acerca de los mismos.

5.1 Contorno Superior e Inferior de un Vehículo

Como se ha dicho, primero hay que partir de la detección para realizar el seguimiento. Así, una vez aplicada la detección de la parte trasera del vehículo se obtiene algo como lo que ya se mostró en la figura 4.8 de la página 55.

Ahora, el primer objetivo es obtener dos puntos que coincidan con la parte superior e inferior del vehículo y estén centrados, es decir, algo como lo presentado en la figura 5.1. para así pintar un segmento vertical con extremos delimitados por estos dos puntos.

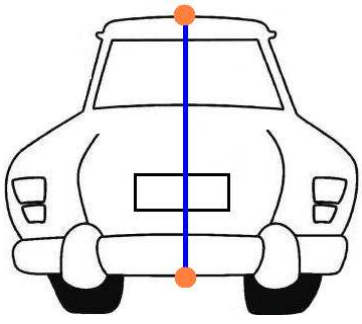

Figura 5.1: Primer paso del seguimiento

La recta es meramente ilustrativa, en el programa estos segmentos se dibujan para comprobar visualmente que el seguimiento funciona adecuadamente, pero no sería necesario que aparecieran en la imagen final mostrada por pantalla. Aún así se dejaran dibujadas dada su alta utilidad didáctica.

Volviendo a lo dicho, para conseguir el objetivo de la figura 5.1, lo primero que se debe hacer es aislar los vehículos presentes en la imagen. Para ello se usarán los rectángulos de las Cascadas de Haar que contienen los objetos de interés y las funciones:

· *cvSetImageROI* que crea una región de interés con cada rectángulo.

· *cvCreateImage* que crea una nueva imagen.

· *cvCopy* que copia la región de interés a esta nueva imagen creada.

Así pues, después de aplicar estas tres funciones de las librerías OpenCV [17] se tendría una nueva imagen con cada vehículo detectado. Para la imagen de la figura 4.8 se obtendrían las tres imágenes reunidas en la figura 5.2.

Figura 5.2: Regiones de interés de la figura 4.8.

5.1.1 Preprocesado de la Imagen

Ahora, para cada región de interés (ROI) de la figura 5.2 se realiza un preprocesado. El objetivo de este pretratamiento es mejorar las imágenes y prepararlas para el paso del apartado 5.1.2: la detección de bordes horizontales mediante el operador Sobel. Consiguiendo así una mayor precisión en la ubicación de los puntos superior e inferior y, por consiguiente, de la recta vertical.

El preprocesado y la aplicación del operador Sobel lo realiza la misma función, *Edges.cpp*, cuyo diagrama de flujo se expone en la siguiente página.

Figura 5.3: Diagrama de flujo de la función Edges.cpp

Como se ve, la función *Edges.cpp* parte de una ROI, la procesa para obtener sus bordes horizontales y, finalmente obtiene los puntos superior e inferior representados en naranja en la figura 5.1.

Las etapas de la figura 5.3 se irán explicando e ilustrando de una en una para que se comprendan fácilmente. Se tomarán las tres regiones de interés de la figura 5.2 para visualizar cómo afecta cada paso a la imagen original.

La primera etapa consiste en una transformación de la ROI en escala de grises. El lector puede pensar que esta manipulación del color es innecesaria dado que la ROI inicial está en blanco y negro pero, como vemos en la figura 5.4, sí que se aprecian cambios notables. Esto es debido a que la función *cvCvtColor* de la librería OpenCV crea una imagen en escala de grises "estandarizada" (por decirlo de algún modo), en otras palabras, este paso es necesario para poder obtener resultados adecuados en los pasos posteriores.

Figura 5.4: Regiones de interés en escala de grises

En la segunda etapa se aplica una gaussiana para suavizar la imagen. Este paso es en realidad un filtrado, pues se usa un filtro paso bajo para eliminar ruido o detalles pequeños de poco interés que sólo afectan a zonas con muchos cambios. De entre todos los posibles filtros paso bajo (media, mediana, promedio, etc.) se eligió la gaussiana pues es el que mejores resultados dio. Los resultados se muestran a continuación:

Figura 5.5: Regiones de interés suavizadas con una gaussiana

Por último, para dejar la imagen preparada para la detección de bordes horizontales mediante el operador Sobel, se aplica una ecualización del histograma. Para ello se ha usado la función *cvEqualizeHist* de la librería OpenCV, obteniendo la figura 5.6.

Figura 5.6: Regiones de interés con el histograma ecualizado

Como se aprecia en la figura anterior, las tres regiones "se ven mejor", esto es debido a que la ecualización del histograma tiene por objetivo transformar una imagen para obtener un histograma con una distribución uniforme. Es decir, que exista el mismo número de píxeles para cada nivel de gris del histograma (ver figura 5.7). Este método es ampliamente utilizado ya que maximiza el contraste de una imagen sin perder información de tipo estructural.

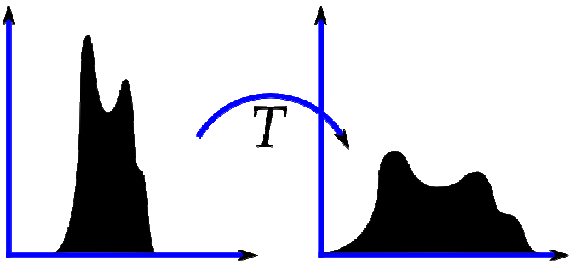

Figura 5.7: Ecualización (no ideal) de un histograma

En resumen, primero se estandariza la imagen en escala de grises, luego se le aplica un filtro paso bajo para eliminar ruido y obtener una imagen más limpia y, finalmente, se manipula su contraste para que la imagen se vea todo lo mejor posible.

Una vez hecho el preprocesado ya se puede pasar a detectar los bordes horizontales de los objetos presentes en la imagen.

5.1.2 Operador Sobel

El operador Sobel [2] [23] calcula el gradiente de la intensidad de una imagen en cada píxel, dando como resultado la magnitud del mayor cambio posible, la dirección de éste y el sentido desde oscuro a claro. El resultado muestra qué tan abruptamente o suavemente cambia una imagen en cada punto analizado y, en consecuencia, qué tan probable es que éste represente un borde en la imagen y, también, la orientación a la que tiende ese borde.

Matemáticamente, el gradiente de la función de intensidad de una imagen es un vector bidimensional cuyas componentes están dadas por las primeras derivadas de las direcciones verticales y horizontales. El operador Sobel utiliza dos kernel de 3x3 elementos, uno para los cambios verticales y otro para los cambios horizontales, que convoluciona a la imagen original para calcular las aproximaciones a estas dos derivadas. Para cada punto de la imagen, el vector gradiente apunta en dirección del máximo incremento posible de la intensidad, y la magnitud del vector gradiente corresponde a la cantidad de cambio de la intensidad en esa dirección.

Lo dicho en los párrafos anteriores implica que el resultado de aplicar el operador Sobel sobre una región con intensidad de imagen constante es un vector cero, y el resultado de aplicarlo en un punto sobre un borde es un vector que cruza el borde (perpendicular) cuyo sentido es de los puntos más oscuros a los más claros.

Por tanto, si únicamente se desea detectar los bordes horizontales de una imagen sólo se deben elegir éstos resultados de aplicar el operador Sobel. Para conseguirlo se utilizará la función *cvSobel(imagen fuente, imagen destino, ordenX, ordenY, apertura)* de la librería OpenCV, con *ordenX = 0* y *ordenY = 1* pues representan los órdenes de las derivadas en la dirección X e Y (y buscamos bordes horizontales) y *apertura = 3* pues es el valor típico usado.

Así pues, si se aplica esta función a la figura 5.6 se obtendrán los resultados de la próxima página, donde el color blanco indica bordes y el color negro zonas homogéneas.

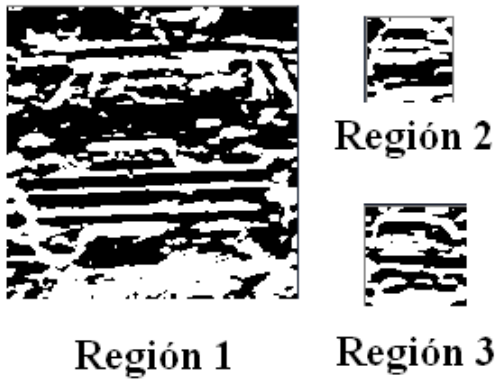

Figura 5.8: Regiones de interés tras aplicar el operador Sobel

Como se ve, el operador Sobel realiza adecuadamente su trabajo y, pese a la complejidad de las formas presentes en las imágenes, se aprecian los bordes horizontales de los objetos resaltados.

Los resultados son correctos, pero el problema es que hay demasiados bordes en la imagen, no se necesitan tantos, sólo se quieren los bordes horizontales correspondientes al contorno del coche, ¿cómo se soluciona este inconveniente? La respuesta es la última etapa en el procesamiento de la imagen, la umbralización.

En nuestro caso, se denomina umbralización al acto de elegir un valor, llamado umbral, a partir del cual todo píxel con un valor superior a ese umbral se considera borde y con valor inferior se considera zona homogénea. Es decir, gracias a la umbralización se pueden conseguir resultados como el mostrado en la figura 5.9.

Figura 5.9: Regiones de interés tras umbralizar (umbral = 100)

Como se ve en la figura 5.9, hay muchos menos bordes, pero los bordes horizontales del contorno superior e inferior se siguen apreciando, por lo que se ha conseguido finalmente un resultado conforme a lo que se buscaba.

Ahora el lector se estará preguntando, ¿cómo se utilizan las imágenes de la figura 5.9 para conseguir el objetivo final buscado? La respuesta a esta pregunta se corresponde con el último paso que realiza la función *Edges.cpp*, y se detalla a continuación.

Primero, hay que recordar que una imagen es, en realidad, una matriz de números, por lo que se puede operar con ella. Estos números van desde 0 hasta 255 siendo el primero correspondiente al color negro y el segundo correspondiente al color blanco. Esta propiedad de las imágenes es la que se utilizará sumando los valores de los píxeles de las regiones de la figura 5.9 fila por fila.

Siendo más precisos, el método implementado en la función *Edges.cpp* consiste en ir tomando las filas de la imagen una a una y sumando los valores que toman los píxeles de cada fila, luego se divide este valor entre 255 para obtener el número de píxeles blancos de cada fila. Así pues, si se representara una gráfica con el número de píxeles blancos en el eje X y el número de la fila en el eje Y se obtendría, generalmente, una distribución como la mostrada en la figura 5.10.

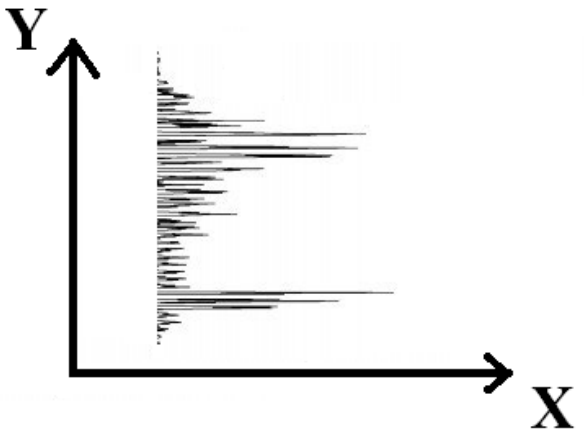

Figura 5.10: Distribución vertical de píxeles blancos

Donde los dos picos máximos corresponderían con el borde superior e inferior del vehículo detectado. Esto permite determinar con bastante precisión los valores que toman las coordenadas Y de los puntos del borde superior e inferior. Para establecer el valor de las coordenadas X bastaría con elegir el de la mitad del ancho del rectángulo de la Cascada de Haar (para que la recta vertical quede centrada). Finalmente, gracias a la función *line* de OpenCV y a estas coordenadas X-Y se podría dibujar un segmento entre estos dos puntos.

Seguidamente se presenta un ejemplo de la aplicación de la función *Edges.cpp* a la figura 4.8.

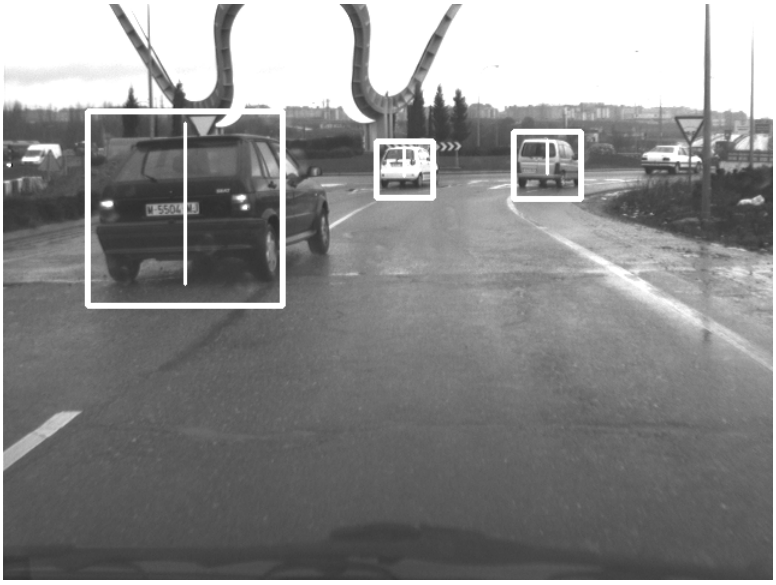

Figura 5.11: Resultado detección de contorno superior e inferior

Y con esto habría concluido el primer paso del seguimiento; en el siguiente apartado se utilizarán las rectas verticales para la extracción y búsqueda de un patrón.

5.2 Búsqueda de un Patrón

Ahora que ya están dibujados los segmentos verticales, el siguiente paso consiste en extraer patrones que nos permitan reconocer cada vehículo distinguiéndolo de los otros y que, también, sean fácilmente localizables en otras imágenes para lograr su seguimiento y, por ende, el de cada vehículo.

Para extraer un patrón de un vehículo se utilizarán las coordenadas de la recta vertical previamente dibujada; con ellas se puede tomar un rectángulo del mismo largo que la citada recta pero algo más grueso, consiguiendo así una nueva ROI larga y delgada. Así pues, para la figura 5.11 se obtendrían tres regiones de interés de donde se extraerían los patrones correspondientes a cada vehículo.

Como se dijo en el subapartado 5.1.2, una imagen es una matriz de números y, por consiguiente, una ROI será una submatriz de números, luego se puede operar con ella al igual que se hizo con las regiones de interés en el subapartado 5.1.1.

Si se le aplica a una de estas delgadas regiones el preprocesado del subapartado 5.1.1 (escala de grises, suavizado y ecualización del histograma) se conseguiría una subimagen lista para ser convertida en una matriz de números, además, al ser de un tamaño tan reducido, estos tres pasos del pretratamiento se harían sin apenas gastar tiempo de procesado.

En resumen, tras todos los pasos citados se habría pasado de tener tres delgados rectángulos coincidentes con las rectas verticales de la figura 5.11, a tener tres pequeñas matrices de números. Ahora, únicamente quedaría sumar los valores de los elementos de cada fila de cada matriz para obtener lo que se estaba buscando: tres vectores columna con números relacionados con cada vehículo.

Estos vectores columna contienen números prácticamente aleatorios y sin ningún tipo de orden, por lo que deben ser modificados para que sean de utilidad. Este paso se hará gracias a la herramienta matemática comentada en el siguiente apartado, el rango intercuartílico.

5.2.1 Rango Intercuartílico

El rango intercuartílico se define como una herramienta matemática del ámbito de la estadística que permite medir la dispersión y variabilidad de una serie de muestras. Para obtener el rango intercuartílico hay que hacer la diferencia entre el tercer y el primer cuartil de una distribución.

Si se tiene una serie de números ordenados de menor a mayor, el primer cuartil se define como la mediana de la primera mitad de estos valores; mientras que el tercer cuartil se calcula determinando la mediana de la segunda mitad de los valores. Por tanto, el segundo cuartil se trata de la propia mediana de la serie de valores.

Se ha elegido el rango intercuartílico como herramienta para extraer un patrón característico pues presenta mejores propiedades que la mediana. Hay una propiedad en concreto que es la que lo hace muy interesante para el seguimiento: su elevada robustez, pues gracias a ella se verá poco perturbado ante cambios o ruido en la imagen y, por tanto, en los datos.

Una vez entendido qué es el rango intercuartílico, lo siguiente es saber cómo se aplica al proyecto. Es bastante sencillo, partiendo de los vectores columna comentados antes, lo primero que se debe hacer es ordenar los números de menor a mayor usando cualquier método de ordenación. Se eligió el algoritmo de la burbuja pues es uno de los más simples y eficientes en cuanto a tiempo de computación. Después de ordenarlos y sabiendo el número de elementos de cada vector, el siguiente paso es dividir el vector por la mitad, en dos grupos de números, para así tomar el primer y el tercer cuartil fácilmente (Q_1 y Q_3). Finalmente, calcular el rango intercuartílico es tan simple como hacer Q_3 menos Q_1.

Así pues, se ha pasado de tener la imagen 5.11 a tener tres números representativos de cada vehículo (los tres rangos intercuartílicos), que se asemejarían con sus “huellas dactilares” por así decirlo.

Por tanto, una vez comprendido todo el procedimiento descrito hasta ahora es fácil suponerse cómo se realiza el seguimiento. Efectivamente, el seguimiento se llevará a cabo realizando este mismo procedimiento en la siguiente imagen de la secuencia pero, además de realizarlo extrayendo un delgado rectángulo coincidente con el nuevo segmento vertical dibujado, también se realizará un pequeño barrido con una ROI de las mismas dimensiones pero que se desplaza por (aproximadamente) toda la región interior del rectángulo de la Cascada de Haar.

En otras palabras, se realizará un barrido por todas las zonas probables donde se podría haber desplazado el vehículo en cuestión para encontrar un nuevo rango intercuartílico muy próximo (o igual) al valor obtenido previamente. Estos rangos intercuartílicos se consiguen procesando una delgada ROI según lo explicado. De todos ellos se elige el más próximo al rango intercuartílico calculado anteriormente. Así, una vez determinado, se dibuja la nueva recta vertical donde se encontró esa ROI y, finalmente se recoloca el rectángulo de la Cascada de Haar procurando que quede centrado tomando este segmento como referencia.

Si realizamos este procedimiento reiteradamente para cada nueva imagen de la secuencia y almacenando el nuevo rango intercuartílico para buscarlo en la siguiente imagen habremos conseguido el objetivo que buscábamos en este capítulo: el seguimiento de un vehículo mediante la búsqueda de un patrón representativo.

5.3 Resultados

En este apartado se muestran los resultados obtenidos con el método expuesto a lo largo del capítulo. Dado que con una única imagen no se puede apreciar el seguimiento, se han reunido en la figura 5.12 un extracto de las imágenes que componen la secuencia completa. El orden en el que deben interpretarse las imágenes de la citada figura es de izquierda a derecha y de arriba a abajo.

Para generar las subimágenes de la figura 5.12 se ha utilizado el método de detección de vehículos comentado en el apartado 4.1: Detección de la Parte Trasera de un Vehículo, es decir, se han utilizado las funciones *DetectAndDraw.cpp*, *Edges.cpp* e *Histograms.cpp* entre otras, para conseguir los objetivos de seguimiento propuestos en el presente capítulo.

El extracto de secuencia de la siguiente página contiene imágenes distintas a las de figuras anteriores. Se ha decidido mostrar estos resultados pues con un único vehículo es más fácil de comprender y visualizar todo lo descrito hasta ahora, como la obtención de la recta vertical, la búsqueda del patrón y la recolocación del rectángulo de la Cascada de Haar utilizando como referencia la citada recta vertical.

Figura 5.12: Resultado del seguimiento de un vehículo

5.4 Comentarios

Como se ve en la figura 5.12, los resultados son correctos, una vez capturado el vehículo no se le pierde en ningún momento, por lo que el objetivo primordial del seguimiento se cumple con éxito.

Si el lector es observador se habrá percatado de que en las subimágenes de la figura 5.12 la recta vertical no va de contorno superior a contorno inferior, sino que por la zona inferior se alarga más de la cuenta. Esto es debido al efecto de la sombra del vehículo, que provoca un cambio de contraste y, por tanto, una falsa clasificación como borde por parte del operador Sobel. Sin embargo, esto no impide que el mecanismo de búsqueda y seguimiento del rango intercuartílico haga correctamente su trabajo. En estas situaciones es cuando sale a relucir la citada robustez de esta herramienta matemática.

Según se dijo al principio del capítulo, sólo se ha implementado un método de seguimiento pese a tener dos procedimientos de detección. Esto es posible gracias a la sencilla adaptabilidad del método, pues, para la detección por partes, sólo se deberían incluir unas pocas líneas de código previas para generar una ROI rectangular que contenga el vehículo entero a partir de las coordenadas de las pequeñas cascadas de las partes. Una vez hecho esto, el preprocesado de la imagen, el dibujo del segmento vertical y la posterior obtención del rango intercuartílico se harían de forma similar.

Capítulo 6

Distancias y Velocidades

Este es el último capítulo perteneciente al núcleo del proyecto, en él se detalla las herramientas usadas para la obtención de distancias y velocidades en imágenes, así como el fundamento teórico en el cual se basan.

Primero se hará una breve introducción a los mapas de disparidad, ya que es necesario conocer qué son y cómo se generan para entender cómo se puede obtener distancias a partir de ellos. Precisamente, a continuación del apartado sobre mapas de disparidad comienza otro sobre obtención de distancias en imágenes. Se comentará más en profundidad en ese apartado pero, a grandes rasgos, la obtención de distancias parte de la creación de un mapa de disparidad gracias a funciones software de las librerías OpenCV y a unas sencillas fórmulas matemáticas que permitirán "traducir" las distancias dentro de la imagen a distancias en la realidad.

También, dentro del apartado 6.2 se trata una funcionalidad adicional que se le ha conseguido extraer a la obtención de distancias mediante mapas de disparidad: la reducción de falsos positivos en el software de detección de vehículos por partes.

Por último, al final del capítulo se explicará cómo obtener velocidades a partir de las distancias calculadas.

6.1 Mapas de Disparidad

Un mapa de disparidad [2] [16] es una imagen donde el nivel de gris en cada píxel indica la distancia horizontal a la que se encuentra el píxel correspondiente de una imagen en la otra. Así pues, para generar un mapa de disparidad son necesarias dos imágenes de una misma escena captadas mediante un sistema de visión estéreo: una imagen izquierda y una imagen derecha.

Los mapas de disparidad son el resultado de la aplicación secuencial de una serie de pasos que enunciamos a continuación:

1. *Calibración*: primero, para obtener buenos mapas de disparidad es necesario calibrar adecuadamente las dos cámaras del sistema estéreo. Este proceso suele hacerse mediante funciones software, en nuestro caso las cámaras han sido correctamente calibradas y, por tanto, las secuencias de imágenes con las que se desarrolla este proyecto proporcionan mapas de disparidad correctos. Otro de los objetivos de la calibración consiste en obtener los parámetros extrínsecos e intrínsecos del sistema. Los que se destacan a continuación son los que se han usado para generar mapas de disparidad:

    a. *Baseline*: es la distancia de separación entre las dos cámaras del sistema estéreo. En nuestro caso toma un valor de 0.119915 metros.

    b. *Distancia focal*: es la distancia entre el eje de la lente y el punto focal (donde se concentran los rayos de luz). Es adimensional y toma un valor de 811.9104 para las cámaras usadas.

    c. *Centro de la Imagen*: mediante la calibración se puede obtener el centro de las imágenes que se van a captar. Para una imagen cualquiera, en nuestro sistema, las coordenadas X-Y del centro son (247.1637, 321.561) píxeles.

2. *Rectificación*: después de calibrar se puede proceder a la aplicación de un método de rectificación para obtener mejores mapas de disparidad. No se ha utilizado ningún método pues no se ha creído necesario.

3. *Correspondencia*: el paso más importante a la hora de generar un mapa de disparidad. Consiste en realizar el cálculo de la correspondencia de un punto de la imagen con la tomada por la otra cámara. Existen dos tipos de algoritmos de correspondencia fundamentales que permiten obtener dos clases de mapas de disparidad:

    a. *Mapas de disparidad densos*: se generan mediante algoritmos basados en el emparejamiento de áreas. Son métodos lentos, pueden tardar desde varios segundos hasta varios minutos por imagen en generar el mapa de disparidad, pero, en cambio, son métodos que permiten obtener resultados muy precisos. Las librerías OpenCV tienen la función *cvFindStereoCorrespondenceGC* para crearlos.

    b. *Mapas de disparidad no densos*: son los mapas de disparidad que se utilizarán en el proyecto. Se consiguen mediante algoritmos que buscan puntos característicos de la imagen, como los bordes o las esquinas, lo que los hace tremendamente rápidos, pudiendo procesar varias imágenes por segundo. El principal problema es que, si los parámetros no se configuran adecuadamente, los resultados que se obtienen son pobres. Las librerías OpenCV tienen la función *cvFindStereoCorrespondenceBM* (utilizada en el proyecto) para crearlos.

4. *Obtención del máximo y de la disparidad*: finalmente, una vez calculada la correspondencia para cada píxel con todos los de la otra imagen, se procede a la elección del punto que mejor se asemeja de todos los candidatos. A partir de esta correlación máxima se obtiene la disparidad y, por tanto, el mapa de disparidades.

En resumen, para que el concepto de mapa de disparidad quede claro, llamamos mapa de disparidad a la imagen que presenta en cada píxel (x, y) el valor de disparidad donde se encontró el mejor valor de correspondencia.

A la hora de representar los valores antes mencionados en la imagen, éstos se suelen normalizar de 0 a 255 para ayudar a percibir mejor las diferencias entre los distintos niveles de gris. También, después de generar el mapa de disparidad, es muy común realizar un filtrado del mismo para mejorarlo. En nuestro caso se ha realizado el filtrado y la posterior normalización mediante las funciones *cvSmooth* y *cvNormalize* de OpenCV.

En la figura 6.1 se muestran las imágenes izquierda y derecha de una secuencia junto al mapa de disparidad que generan.

Figura 6.1: Ejemplo de mapa de disparidad

6.2 Obtención de Distancias

Una vez comprendidos los mapas de disparidad y cómo se obtienen, lo siguiente es saber cómo deben ser utilizados para determinar el valor de la distancia de un objeto detectado mediante Cascadas de Haar.

Partiendo de la figura 6.1, si se le aplica la detección de la zona trasera de un vehículo mediante Cascadas de Haar (tal y como se explicó en el apartado 4.1) a la imagen derecha se obtendrá algo similar a la figura 6.2.

Figura 6.2: Ejemplo detección mediante Cascadas de Haar

Ahora bien, se tiene el mapa de disparidad de la figura 6.1 y la imagen resultante de la figura 6.2, ¿cómo se determinan las distancias?

El proceso es sencillo, lo primero es determinar las coordenadas X-Y de un punto cualquiera en la imagen 6.2 del cual se quiera conocer la distancia real. Según la finalidad de este apartado, lo que se desea saber es la distancia a la que nos encontramos del vehículo detectado, así pues, el punto representativo del vehículo que se ha elegido es el punto medio del segmento vertical que va de contorno superior a contorno inferior. Con estas coordenadas y el mapa de disparidad lo siguiente es obtener el valor que toma este punto en el mapa de disparidades pero, como se dijo en el apartado anterior, este valor no nos da la distancia en unidades métricas, sino que se necesita una ecuación matemática que nos "traduzca" un valor adimensional en otro con unidades de metros. Esta ecuación es la siguiente:

$$Z = \frac{f \cdot B}{val_1}$$

Donde f es la distancia focal de la cámara (811.9104), B es el *baseline* (0.119915 metros) y val_1 es el valor que se extrae del mapa de disparidad dividido por la resolución que, en nuestro caso, toma un valor de 16.

Así pues, si aplicamos esta fórmula al píxel central del segmento se obtiene la distancia Z en metros a la cual se encuentra. También se pueden calcular otras dos distancias que pueden ser de utilidad mediante unas fórmulas análogas, las distancias X e Y del objeto con respecto al centro de la imagen.

$$X = \frac{Z \cdot u}{f} \qquad Y = \frac{Z \cdot v}{f}$$

Donde Z es la distancia en metros calculada previamente, f es la distancia focal, "u" es la diferencia entre la columna que ocupa el píxel dentro de la imagen y la columna que ocupa el centro de la imagen, y "v" la diferencia entre la fila que ocupa el píxel dentro de la imagen y la fila que ocupa el centro de la imagen.

Con todo lo dicho, lo siguiente que queda es implementar una función en lenguaje C++ con la que conseguir distancias (ver *CarsDetect.cpp* y *Distances.cpp* en el Apéndice de este proyecto). Para la imagen estéreo de figuras anteriores se han obtenido los valores que se aprecian en la figura 6.3.

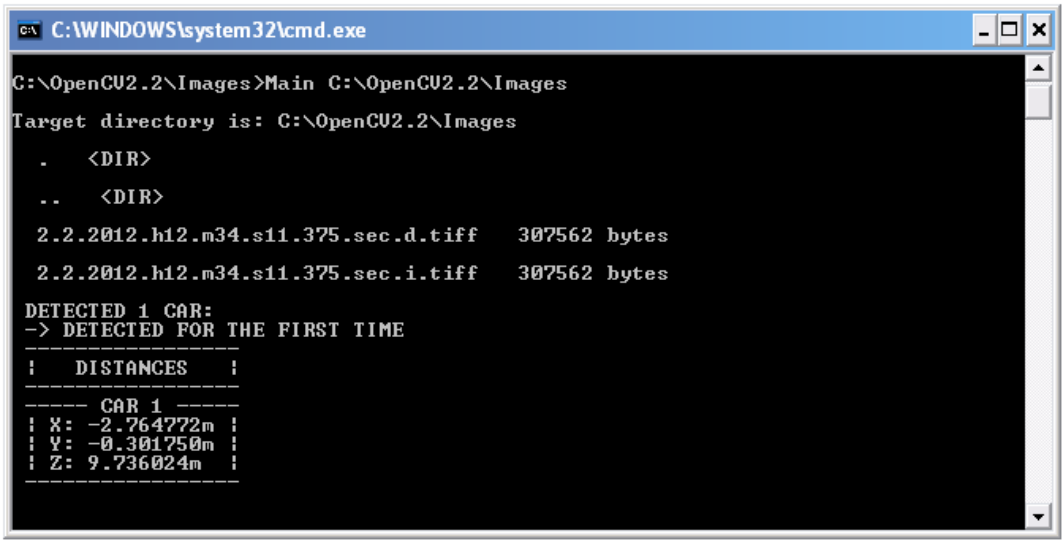

Figura 6.3: Resultado cálculo de distancias

Finalmente comentar que, para el segundo método de detección de vehículos desarrollado en este proyecto, el método de detección por partes, el cálculo de distancias se usó también para mejorar su precisión. Esta reducción de falsos positivos se consiguió llevar a cabo de la siguiente forma:

- Partimos de las agrupaciones de las partes obtenidas y del conocimiento de cuales de los siete posibles casos distintos (ver figura 4.16, página 65) se han detectado.
- Para cada caso detectado, se toman los vértices de las Cascadas de Haar marcados con una X en la figura 6.4.
- Se calcula el valor de la disparidad para cada uno de estos vértices y se comprueba que ruedas y esquinas superiores tienen el mismo valor de disparidad. Aquellos grupos que no cumplan esto serán descartados pues esto querrá decir que sus partes tienen distintos niveles de profundidad.

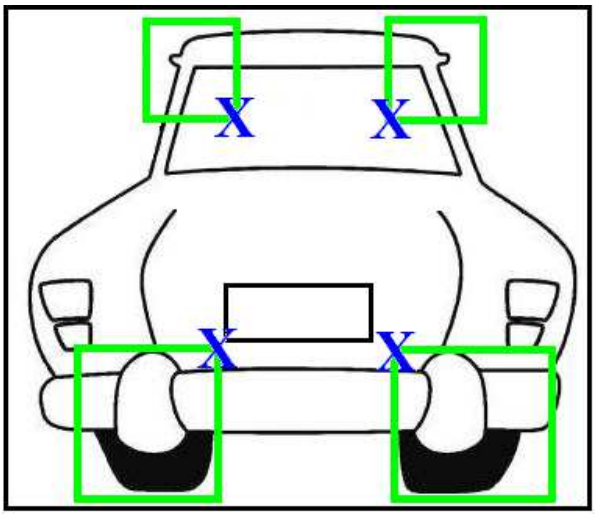

Figura 6.4: Vértices analizados en el cálculo de distancias

Como última anotación decir que, para los casos 6 y 7 donde únicamente hay una esquina, una rueda y la matrícula, se tomó por ejemplo la esquina superior izquierda de la matrícula para calcular la disparidad pues era indiferente la que eligiéramos. En los otros casos se ha obviado usar la matrícula pues con la disparidad de esquinas y ruedas se dispone de información suficiente.

6.3 Obtención de Velocidades

La velocidad de un objeto cualquiera se calcula como el lector bien sabe, dividiendo el espacio recorrido por el objeto entre el tiempo empleado en recorrerlo. Así pues, esta segunda parte del capítulo es inmediata, ya que la velocidad del vehículo detectado se calcula a partir de la distancia previamente obtenida.

Por tanto, para un par de imágenes consecutivas de una secuencia, la velocidad se calcula dividiendo el incremento de distancia entre el tiempo que transcurre entre foto y foto. Este valor temporal es sencillo de calcular pues el sistema de visión estéreo del IVVI, tras capturar una imagen, la guarda como archivo .tiff con el siguiente nombre:

Día.Mes.Año.Hora.Minutos.Segundos.Secuencia.tiff

Donde la secuencia nos dice si la imagen corresponde a la cámara derecha o a la izquierda. Un ejemplo de nombre de imagen sería:

2.2.2012.h12.m34.s11.375.sec.d.tiff

Así pues, como la variación entre foto y foto es del orden de décimas de segundo, con los tres últimos números del nombre de la imagen se conseguiría la información temporal buscada.

Figura 6.5: Imágenes de ejemplo para el cálculo de velocidades

En la figura 6.6 se muestra un resultado práctico del valor de la velocidad que aparecería por pantalla para las imágenes de la figura 6.5.

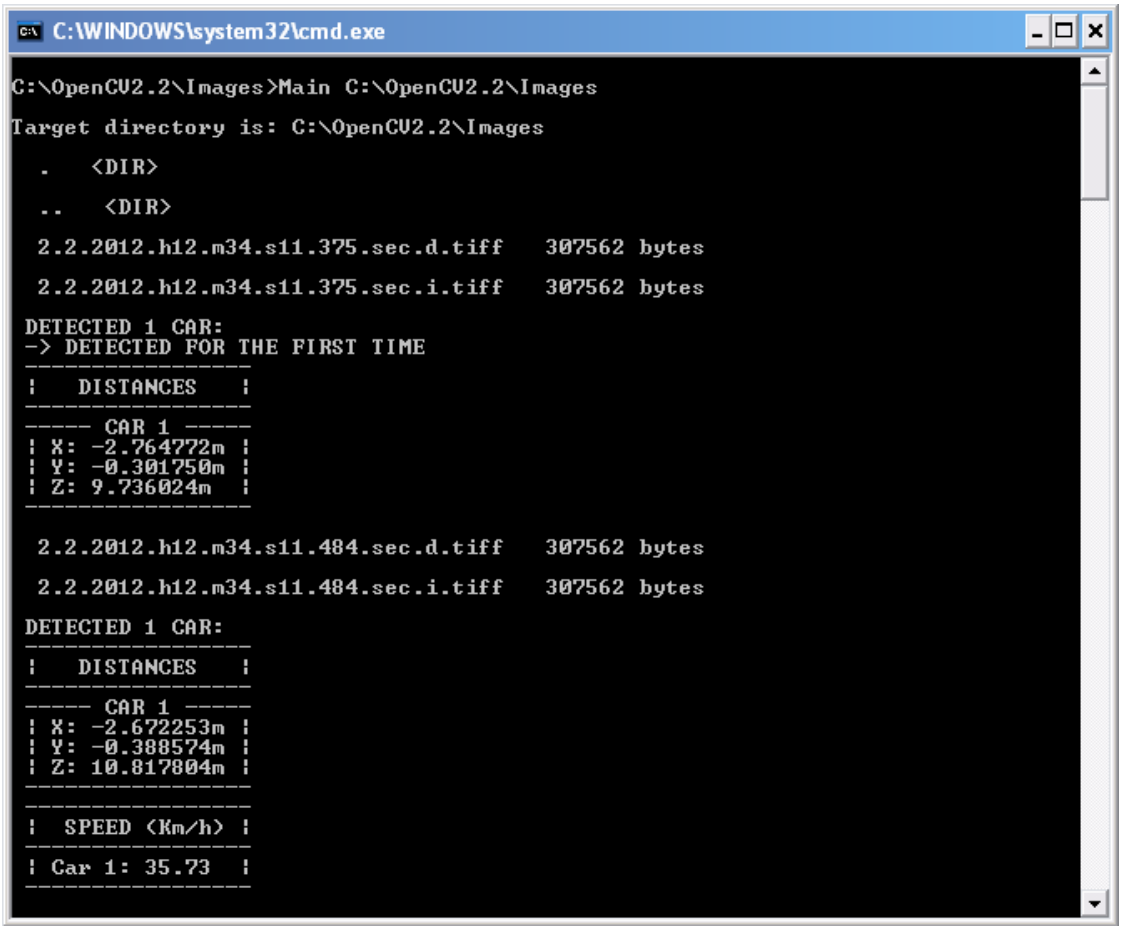

Figura 6.6: Resultado cálculo de velocidades

Como se ve, la velocidad que finalmente se obtiene es de 35.73Km/h. El lector, si es perspicaz, se habrá percatado de que, de hecho, esta no es la velocidad real del vehículo que se ha detectado, pues nuestro vehículo IVVI también se está desplazando a una determinada velocidad y, por tanto, la velocidad que se muestra en la figura 6.6 no es sino la velocidad relativa del vehículo.

Si se desea saber la velocidad absoluta a la que se está desplazando el vehículo, se debe sumar el valor de nuestra velocidad al valor de velocidad relativa calculado. En los programas desarrollados se ha optado por calcular únicamente velocidades relativas pues, a efectos prácticos, es indiferente una u otra ya que sólo se diferencian por la adición de una constante.

Capítulo 7

Conclusiones

Una vez explicado el objetivo de este proyecto, así como todas las partes del mismo, lo siguiente es hacer una prueba de los programas implementados y una posterior valoración de los resultados obtenidos. Antes de esto, se ha creído conveniente exponer un breve resumen de todo lo comentado hasta ahora, sin entrar en mucho detalle, pero lo suficientemente explícito para así tener una visión completa de todo el trabajo desarrollado.

Por tanto, en este capítulo, primero se expondrá una visión global del proyecto, a modo ilustrativo, para entender cómo acoplar los distintos módulos explicados en secciones previas y así conseguir dos aplicaciones software con el mismo fin pero con diferentes resultados, debilidades y fortalezas.

Consecutivamente, en el siguiente apartado, se mostrarán estas similitudes y diferencias para que el lector entienda el por qué del desarrollo de dos aplicaciones y el granito de arena que aportan cada una al mundo científico.

Por último, al final del capítulo se hará una reflexión personal sobre el trabajo realizado, así como una valoración crítica y objetiva, destacando puntos fuertes y limitaciones de los softwares.

7.1 Visión Global del Proyecto

En capítulos anteriores se ha explicado cada uno de los módulos o partes relevantes de los softwares por separado, ahondando en sus bases teóricas o principios matemáticos, pero en ningún momento se ha explicado cómo se combinan, cómo se relacionan y en qué orden se llama a las diversas funciones; por lo que este apartado es necesario para una mejor comprensión de lo expuesto hasta el momento.

Como se dijo antes, finalmente se han implementado dos aplicaciones software, por lo que se explicará cada una por separado:

- ***Programa 1 – Basado en una Cascada de Haar***

Este programa, como el segundo, hace uso de las Cascadas de Haar tan mencionadas a lo largo de esta memoria, la diferencia estriba en que el programa número 1 utiliza únicamente una Cascada de Haar entrenada para reconocer partes traseras de vehículos, mientras que el programa 2 emplea varias cascadas para la detección por partes del vehículo (ver apartados 4.1 y 4.2).

Así pues, el programa 1 adquiere una imagen de la secuencia y la procesa mediante una Cascada de Haar para buscar los vehículos presentes. Pueden darse varios casos:

- *Que no se detecte ningún vehículo y que en imágenes previas tampoco se detectaran vehículos*: no se tiene información previa y no se ha detectado nada, así que se pasa a la siguiente imagen para realizar una nueva búsqueda.

- *Que se detecte un vehículo y no haya información previa*: en este caso se está detectando un vehículo por primera vez o un falso positivo, por lo que se debe procesar la información adecuadamente y almacenar su rango intercuartílico (ver apartado 5.2.1) para el posterior seguimiento en caso de que la detección con memoria (ver apartado 4.3) nos ratifique que se trata de un objeto de interés y no de un error. Por último se calculará la distancia del objeto detectado (ver apartado 6.2).

- *Que se detecte un vehículo y haya información previa*: nos encontramos ante el caso más común, que se haya detectado un vehículo previamente y que ahora se siga detectando. Se debe actualizar la información para el seguimiento, calcular la distancia y, por último, calcular la velocidad a la que se desplaza, pues se conocen las distancias y tiempos de imágenes consecutivas (ver apartado 6.3).

El diagrama de flujo que resume el comportamiento de este primer programa se muestra en la figura 7.1.

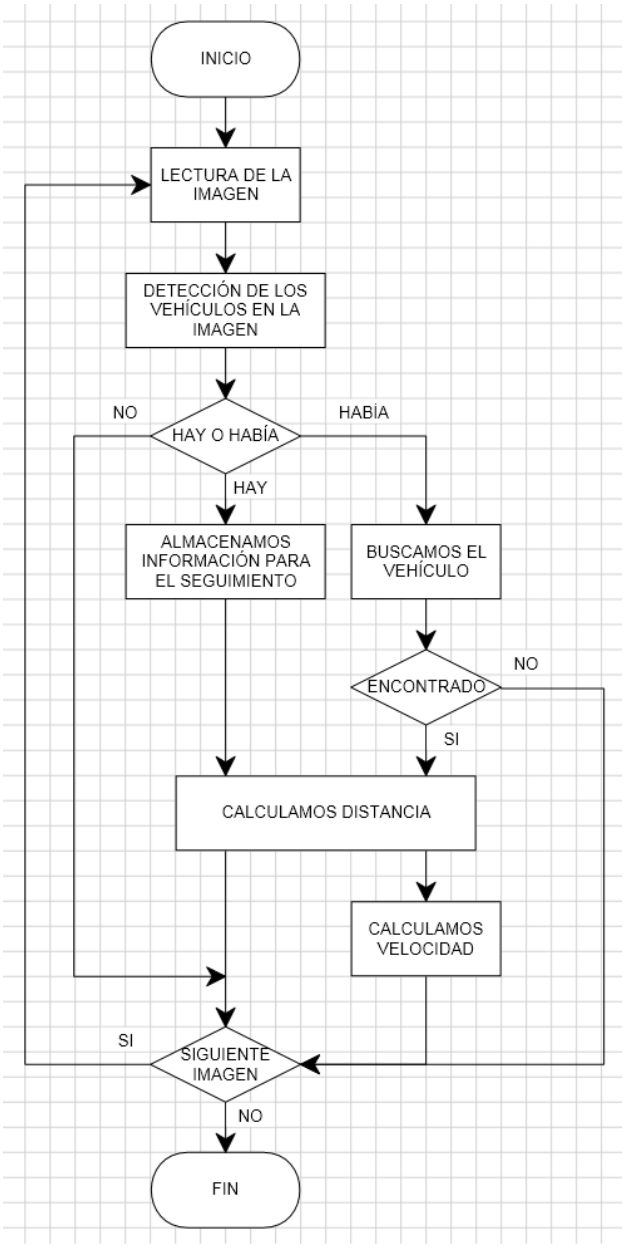

Figura 7.1: Diagrama de flujo del programa 1

- ***Programa 2 – Basado en múltiples Cascadas de Haar***

El segundo programa tiene el siguiente diagrama de flujo:

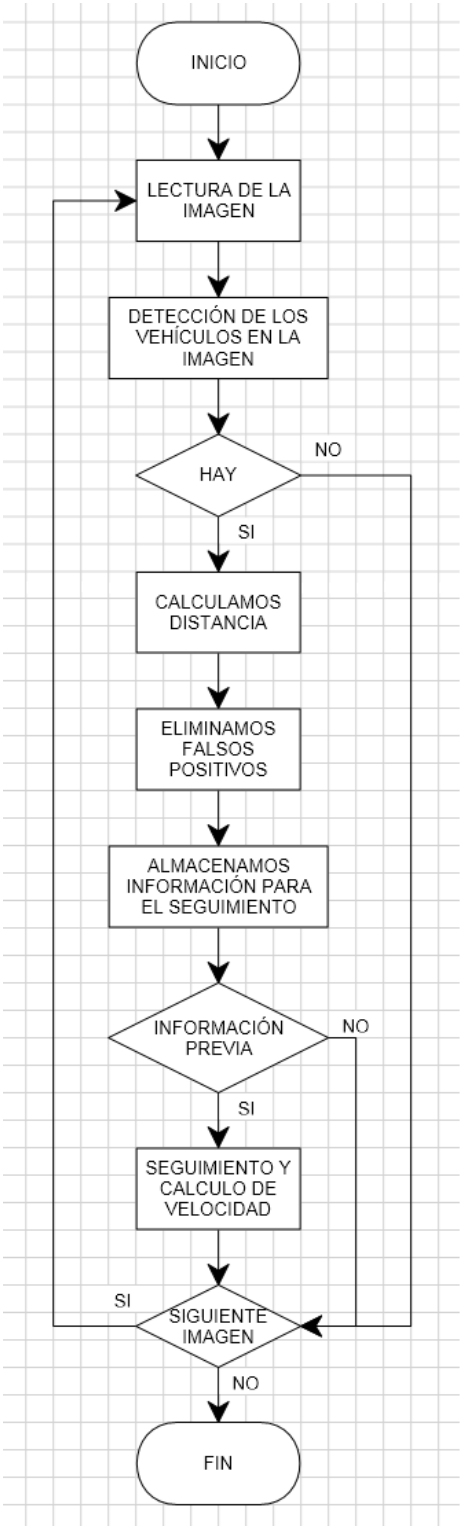

Figura 7.2: Diagrama de flujo del programa 2

Viendo la figura 7.2, este segundo programa tiene bastantes diferencias con respecto al ya comentado programa 1. El cambio más notable es que el diagrama de flujo, en general, es más simple que el de la figura 7.1. Esto es así pues, como se dijo en el apartado 4.3, la detección con memoria únicamente se iba a implementar para el programa 1 debido a problemas de complejidad. También hay que destacar que, en contraste, en este programa 2 el seguimiento se realiza a continuación de la obtención de distancias; se ha escogido este orden pues el cálculo de distancias permite eliminar bastantes falsos positivos (ver apartado 6.2) y, por tanto, el seguimiento será más preciso si se realiza a posteriori.

Aún a pesar de estas diferencias, el objetivo es el mismo, detectar los vehículos presentes en la imagen, calcular a qué distancia están y a qué velocidad se mueven y, por último, guardar información relacionada con el seguimiento para buscar los vehículos en imágenes posteriores.

Por último decir que, si el lector está interesado, al final del presente documento se anexan los códigos en lenguaje C++ que componen cada una de las aplicaciones explicadas.

7.2 Resultados Experimentales

En este apartado se muestran los resultados de ejecutar ambos programas en las mismas secuencias de imágenes para que resulte más ilustrativa la comparativa.

Las siguientes figuras muestran a la izquierda el resultado del programa 1 y a la derecha el del programa 2 para esa imagen de la secuencia. A pie de foto se escriben las distancias y velocidades obtenidas (en caso de haberlas) con cada programa, los subíndices 1 y 2 hacen referencia al programa 1 y 2 respectivamente.

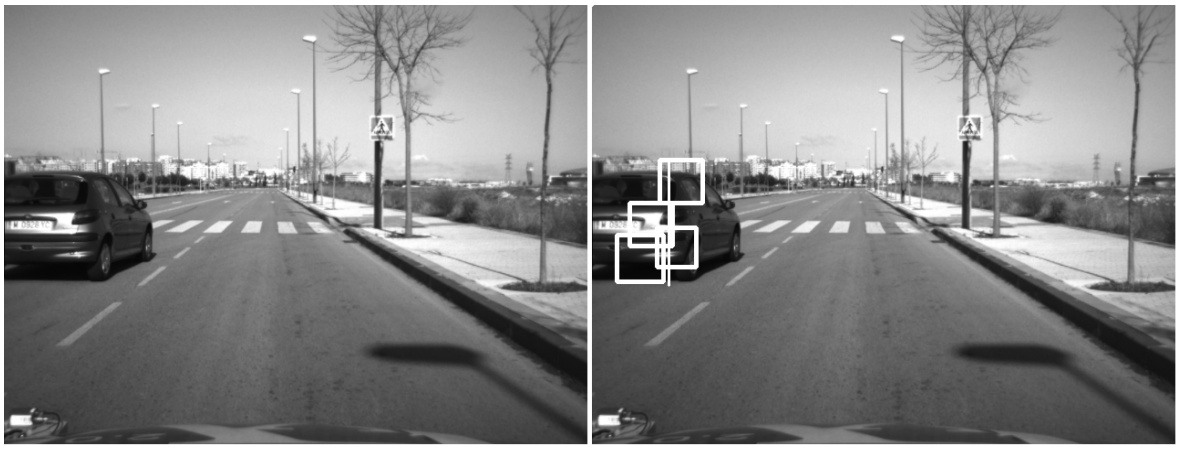

Figura 7.3: Imagen 1 – Z_2: 8.851m

Figura 7.4: Imagen 2 – Z_1: 8.851m, Z_2: 9.736m, V_2: 28.97Km/h

Figura 7.5: Imagen 3 – Z_1: 9.736m, V_1: 25.49Km/h

Figura 7.6: Imagen 4 – Z_1: 9.736m, V_1: 25.49Km/h

Figura 7.7: Imagen 5 – Z_1: 10.818m, Z_2: 10.818m, V_1: 35.73Km/h

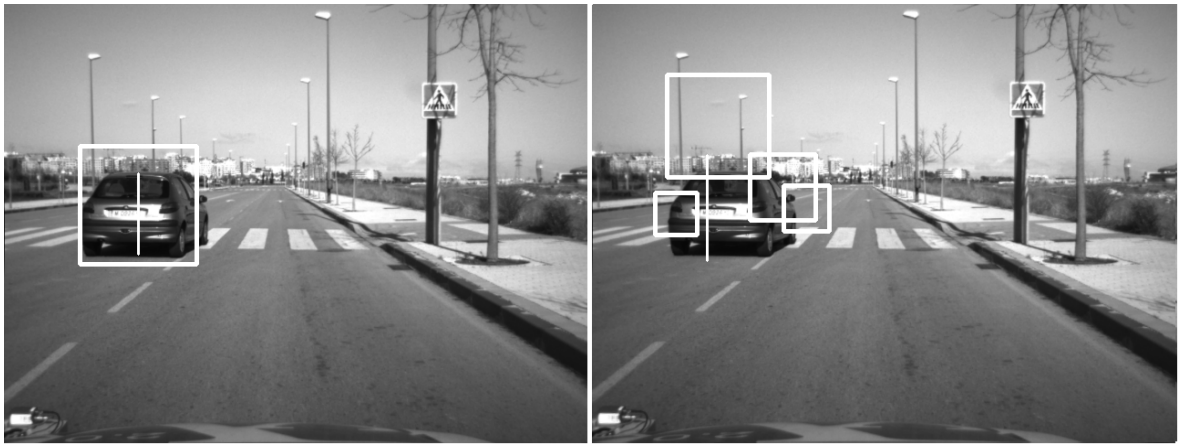

Figura 7.8: Imagen 6 – Z_1: 10.818m, Z_2: 12.17m, V_1: 35.73Km/h, V_2: 34.52Km/h

7.3 Comentarios Finales y Valoración Objetiva

A la vista de las seis figuras anteriores, y con sólo esta información, se podría deducir que el programa 1 es mejor que el programa 2, esto es cierto, pero sólo en parte.

El programa 1 es más preciso, es decir, detecta más, esto se debe a que el programa 2, a parte de utilizar varias cascadas que buscan objetos más pequeños y, por tanto, más complicados de encontrar, presenta condiciones adicionales de restricción espacial para agrupar las partes detectadas. Estás dos características, detectar las partes del vehículo y que se encuentren dentro de un rango de distancias y profundidad establecidos, deben cumplirse simultáneamente y, por tanto, en muchas ocasiones el resultado será una imagen limpia, sin ninguna Cascada de Haar.

También, en términos de eficiencia, el programa 1 es mejor ya que el procesado completo de la imagen se hace en un par de segundos a lo sumo, mientras que el programa 2 tarda aproximadamente siete segundos en procesar una única imagen. Esto es comprensible si pensamos en el número de Cascadas de Haar empleadas en cada programa, una frente a tres, pues el proceso de comparación de la imagen con la cascada es el que más tiempo requiere.

Una de las ventajas de usar el programa 2 es que es más robusto, en otras palabras, cuando se encuentra un grupo de partes que cumple las restricciones, la probabilidad de que esa “nube de cuadrados” esté superpuesta a un vehículo es muy alta. En las figuras mostradas no aparece, pero el programa 1 ha dado varios falsos positivos en imágenes previas y posteriores al extracto de secuencia mostrado en las seis figuras anteriores. Esto se debe al uso de una única cascada, de la que se depende para el correcto funcionamiento del programa.

Enlazando con lo dicho en el párrafo anterior, como se observa, de las tres cascadas usadas en el programa 2, la que peor funciona es la encargada de detectar ruedas y, en consecuencia, puede ser la causante de que el resultado final no sea el adecuado.

En conclusión, si se desea un programa que funcione bien y cumpla con su objetivo de manera rápida y eficiente, el lector deberá optar por el programa 1. Este es, sin embargo, el método clásico de detección de objetos y, por tanto, tiene pocas posibilidades de modificación para incluir funciones adicionales. En cambio, el programa 2, al ser más experimental e innovador permite nuevas e interesantes funcionalidades. Es cierto que primero requiere unas cuantas mejoras como la ya mencionada cascada de detección de ruedas o una mejora de reducción temporal añadiendo, por ejemplo, que la comparación de la imagen con las Cascadas de Haar sólo se realice en ciertas zonas probables de la imagen una vez detectado el vehículo, etc. Pero, si lo que se busca es un programa con muchas posibilidades y que detecte de manera muy fiable un objeto, la opción debe ser el programa 2. No obstante, como se ha dicho antes, si se desea un programa que funcione relativamente bien y que esté listo para ser usado, se debe escoger, sin duda alguna, el programa 1.

Así pues, en función de cómo sea el lector o la persona interesada en ambos programas, se debe elegir uno u otro.

Capítulo 8

Aplicaciones del Proyecto

Los programas software desarrollados tienen un carácter eminentemente práctico y experimental y, en consecuencia, las aplicaciones que se pueden implementar son muy variadas; sin embargo, todas tienen un elemento común que las unifica: un sistema de visión estéreo que capte el entorno y provea de imágenes a ambos programas.

En este breve capítulo se verá desde la aplicación más inmediata del proyecto y para la cual ha sido desarrollado, hasta el amplio abanico de posibles aplicaciones relacionadas, pasando por otras que, con una breve modificación del código base, podrían llevarse a cabo.

8.1 Aplicaciones Actuales

La aplicación más importante y, a su vez, la que ha motivado el desarrollo de este proyecto es la detección, seguimiento, cálculo de distancias y velocidades de vehículos que circulan por el mismo carril viario que nuestro turismo equipado con un sistema de visión estéreo y que procesa imágenes mediante Cascadas de Haar. El vehículo dotado de visión estéreo no es otro que el ya mencionado IVVI 2.0 [22]. Este coche, el Nissan Note, ha sido modificado por el departamento de Ingeniería de Sistemas y Automática de la Universidad Carlos III de Madrid dotándole de varios sistemas, entre ellos destaca el sistema de visión estéreo mediante cámaras Hitachi blanco y negro de barrido progresivo.

Así pues, es gracias a este sistema estéreo que ambos programas reciben imágenes para ser procesadas. En conjunto, este sistema y los otros de los que ya consta el IVVI buscan como objetivo conseguir un vehículo inteligente que ofrezca un conjunto de servicios y ayudas a la conducción. Los otros sistemas de los que ya consta el IVVI son:

- *Detección de los límites de la carretera y de los carriles*: las cámaras captan imágenes de los carriles que son posteriormente procesadas por el ordenador central del IVVI. Luego, mediante software, se detecta el tipo de carril del que se trata para poder actuar en consecuencia. Reconoce, por ejemplo, líneas continuas (donde no se puede adelantar), discontinuas, y carriles de aceleración. Con esta detección se pretende anticipar el peligro que ocurre cuando el conductor se va alejando inconscientemente del carril.

- *Detección y reconocimiento de las señales de tráfico*: por su forma y color, y gracias a otros algoritmos de búsqueda similares a las Cascadas de Haar, se consigue el reconocimiento de las señales de tráfico. Entre otras aplicaciones como la actuación de los frenos ante una señal de STOP, etc. destaca la posibilidad de crear un vehículo que inspeccione el estado de las señales de tráfico de forma automática, atendiendo a su color, tamaño y/o posición.

- *Detección de peatones*: con las cámaras de detección de señales y, mediante otro software de búsqueda de peatones, se puede lograr avisar al conductor de peatones que puedan suponer un peligro o frenar automáticamente en caso de la no reacción del conductor. También, gracias a otras dos cámaras del espectro infrarrojo se pueden detectar peatones en condiciones de baja visibilidad.

- *Detección de somnolencia en el conductor*: mediante cámaras situadas en el interior del vehículo y enfocadas al rostro del conductor se reconoce cuando el mismo está comenzando a quedarse dormido midiendo la frecuencia de parpadeo. También se ha implementado un reconocimiento de distracciones (el conductor desvía por mucho tiempo la mirada de la carretera), en ambos casos se alerta al conductor de forma sonora.

Así pues, el ayudar a ampliar las prestaciones del vehículo inteligente IVVI es la aplicación fundamental que ha motivado la creación del proyecto pero, existen otras, también interesantes, que se mencionan a continuación.

- *Vehículos policiales*: gracias a las aplicaciones desarrolladas se podría dotar a los vehículos de la policía de la capacidad para detectar infractores que superen el límite de velocidad o, adicionando una lectura de matrículas al programa 2, se podría llevar a cabo la identificación de vehículos robados.

- *Robots móviles*: en el ámbito de la robótica las aplicaciones podrían ser muy variadas, para los robots humanoides la detección, seguimiento, cálculo de distancias y velocidades de otro objeto que no fuera un vehículo, se podría implementar simplemente cambiando la Cascada de Haar por otra entrenada para reconocer pelotas rojas por ejemplo. También, para robots no humanoides que trabajan en equipo o en enjambre como los microrobots, estas aplicaciones de visión artificial podrían resultar interesantes para comunicarse entre ellos o guiarse.

8.2 Repercusiones Futuras

Las modificaciones más inmediatas que añadirían nuevas funcionalidades a las aplicaciones desarrolladas son:

- *Lectura de matrículas*: mediante detectores como Sobel, el detector de Canny o métodos más refinados de extracción de características como SURF de OpenCV se podrían reconocer matrículas para después buscarlas en una base de datos lo que, como se comentó en el apartado 8.1, proporcionaría un útil método de identificación de vehículos robados.

- *Detección de aceleraciones y frenadas de otros vehículos*: mediante un sistema de visión a color y mediante un algoritmo de reconocimiento de faros traseros se podría saber cuándo un vehículo está frenando (paso de rojo oscuro a rojo claro) o acerando (paso de rojo claro a rojo oscuro).

- *Detección de vehículos en sentido contrario*: usando de nuevo las Cascadas de Haar pero esta vez entrenadas para reconocer frontales de vehículos se podrían detectar vehículos en sentido contrario. Una primera aplicación que justifica su desarrollo sería la de evitar colisiones frontales entre vehículos pero, enlazando con el punto anterior, también serviría para detectar e identificar matrículas de vehículos que circulan por el otro carril viario.

Por último, cabe destacar que todas las aplicaciones, a excepción de las relacionadas con el ámbito de la robótica, tienen como objetivo alcanzar el vehículo autónomo o vehículo dotado de piloto automático, tan deseado por el ser humano, para así reducir en gran medida los accidentes de tráfico que tantas vidas siegan actualmente.

Capítulo 9

Presupuesto

En este último capítulo de la presente memoria se calculan, de forma aproximada, los costes totales que conlleva el desarrollo del proyecto. Para realizar la estimación monetaria se ha optado por determinar por separado costes fijos y costes variables, calculando estos últimos a partir de los parámetros temporales de las tablas 9.1 y 9.2.

ETAPA	TIEMPO (horas)
Estudio previo y comprensión del trabajo en su conjunto	10
Comprensión de las herramientas y las bases teóricas	10
Instalación de herramientas software y puesta a punto	5
Implementación de las aplicaciones en código C++	250
Realización de pruebas para verificación	10
Corrección de errores y depurado de los códigos	15
Realización de pruebas finales	10
TOTAL	**310**

Tabla 9.1: Tiempo invertido en el desarrollo de las aplicaciones

ETAPA	TIEMPO (horas)
Estructuración del documento	5
Elección de formatos	5
Búsqueda de información adicional a la memoria	15
Redacción de la memoria	150
Relectura y corrección de errores	5
TOTAL	**180**

Tabla 9.2: Tiempo invertido en el desarrollo de la memoria

Así pues, los costes variables, suponiendo un sueldo de 15€/hora para un ingeniero de software son en total:

$$CV = 15\frac{€}{h}\cdot(310+180)h = 7350€$$

Los costes fijos se muestran en la tabla-resumen 9.3.

CONCEPTO	COSTE (€)
PC de sobremesa (torre y monitor)	1000
Licencia Sistema Operativo Microsoft Windows 7	200
Licencia software (Visual C++ y librerías OpenCV)	0
Accesorios informáticos (impresora, ratón, teclado, etc.)	150
Útiles desechables (papel, lápiz, cartuchos de tinta, etc.)	100
TOTAL	**1450**

Tabla 9.3: Costes fijos del proyecto

Finalmente, los costes totales del proyecto suman una cantidad de:

$$\textbf{Costes Totales} = CF + CV = 1450€ + 7350€ = \textbf{8800€}$$

Bibliografía

[1] Axis Communications: Técnicas de Barrido de Imágenes. Available from: <http://www.axis.com/es/products/video/camera/progressive_scan.htm> [April 2012]

[2] Bradski, G. & Kaehler, A., 2008. Learning OpenCV: Computer Vision with the OpenCV Library. California: O´Reilly.

[3] Cplusplus. Available from: <http://www.cplusplus.com> [March 2012]

[4] DaniWeb: Convert wchar_t* to char*. Available from: <http://www.daniweb.com/software-development/cpp/threads/87362/how-to-convert-wchar_t-to-char> [March 2012]

[5] Diagramas de Flujo, Herramienta de Diseño. Available from: <https://grapholite.com/Designer/> [May 2012]

[6] Freund, Y. & Schapire, R. E., 1996. Experiments with a New Boosting Algorithm. *In: Machine Learning: Proceedings of the 13th International Conference*. 148-156.

[7] González, J., 2011. Cascadas de Haar Aplicadas a Detección de Vehículos. Universidad Carlos III de Madrid.

[8] Gribble, C., 2000. A Brief History of C++. Available from: <http://www.hitmill.com/programming/cpp/cppHistory.html> [April 2012]

[9] Henao, D., 2010. Inteligencia Artificial. Available from: <http://www.monografias.com/trabajos12/inteartf/inteartf.shtml> [April 2012]

[10] Highgui Reference Manual. Available from: <http://www.ai.rug.nl/vakinformatie/pas/content/Highgui/opencvref_highgui.htm> [March 2012]

[11] Lienhart, R. & Maydt, J., 2002. An extended Set of Haar-like Features for Rapid Object Detection. *In: IEEE Conference on Image Processing (ICIP´02), September 2002 New York*. 155-162.

[12] MasterASP: Sistema de Gestión y Cobro para Parking. Available from: <http://www.masterasp.com/resources/contenidos/servmat4c.JPG > [April 2012]

[13] Microsoft Developer Network: Listing the Files in a Directory. Available from: <http://msdn.microsoft.com/en-us/library/aa365200(v=vs.85).aspx> [March 2012]

[14] Nüchter, A., Lingemann, K. & Hertzberg, J., 2005. Gentle Adaboost for CARTs. Available from: <http://kos.informatik.uni-osnabrueck.de/download/icar2005_1/node11.html> [April 2012]

[15] OpenCV Reference. Available from: <http://opencv.willowgarage.com/documentation/c/index.html> [March 2012]

[16] Peris, M., 2011. Stereo Matching. Available from: <http://blog.martinperis.com/2011/08/opencv-stereo-matching.html> [June 2012]

[17] Ruddin, N., 2009. OpenCV Region of Interest. Available from: <http://nashruddin.com/OpenCV_Region_of_Interest_(ROI)> [April 2012]

[18] Seo, N., 2008. Imageclipper. Available from: <http://code.google.com/p/imageclipper> [March 2012]

[19] Seo, N. OpenCV Matrix Operations. Available from: <http://note.sonots.com/OpenCV/MatrixOperations.html> [March 2012]

[20] Sinha, U., 2010. Drawing Histograms in OpenCV. Available from: <http://www.aishack.in/2010/07/drawing-histograms-in-opencv/> [April 2012]

[21] Tecnocarreteras: Sistema Birdwatch. Available from: <http://www.tecnocarreteras.es/web/items/1/270/birdwatch-un-detector-de-vehiculos-que-se-saltan-el-semaforo-en-rojo-muy-facil-de-implantar> [April 2012]

[22] Universidad Carlos III, Sistemas Inteligentes de Transporte: IVVI 2.0. Available from:<http://www.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/lab_sist_inteligentes/sis_int_transporte/vehiculos/IvvI20/> [April 2012]

[23] Universidad de Jaén, 2005. Detección de Bordes en una Imagen. Available from: <http://www4.ujaen.es/~satorres/practicas/practica3_vc.pdf> [April 2012]

[24] Viola, P. & Jones, M. J., 2004. Robust real-time face detection. *International Journal of Computer Vision*, 57 (2), 137-154.

[25] Wikipedia: Cámara Estereoscópica. Available from: <http://es.wikipedia.org/wiki/C%C3%A1mara_estereosc%C3%B3pica> [April 2012]

[26] Wikipedia: Charged-Coupled Device. Available from: <http://es.wikipedia.org/wiki/Charge-coupled_device> [April 2012]

[27] Wikipedia: Visual C++. Available from: <http://es.wikipedia.org/wiki/Visual_C%2B%2B> [April 2012]

[28] Wilson, D., 2012. Detección de Peatones mediante Sistemas de Visión Artificial. Available from: <http://blog.infaimon.com/2012/06/deteccion-de-peatones-mediante-sistemas-de-vision-artificial/> [April 2012]

Apéndice

En esta sección se comentan características importantes del proyecto a nivel informático, como pueden ser los códigos en lenguaje C++ que componen las funciones de ambas aplicaciones, cómo se configura Visual C++ para trabajar con las librerías OpenCV y otros datos de interés, como que se debe situar el ejecutable *Main.exe* y las cascadas *xml* en el mismo directorio que las imágenes, y que el nombre de ese directorio no puede contener espacios.

Primero se expondrán brevemente los pasos a seguir para configurar adecuadamente el entorno de desarrollo Visual C++ por si el lector está interesado en el mundo de la programación y, en concreto, en la creación de aplicaciones de Visión Artificial.

1. *Descargar librerías OpenCV*: pueden descargarse gratuitamente desde la web <sourceforge.net/projects/opencvlibrary>

2. *Instalar las librerías OpenCV*: se debe seleccionar la opción de la figura A1 y elegir como directorio de instalación "`C:/OpenCV2.2`"

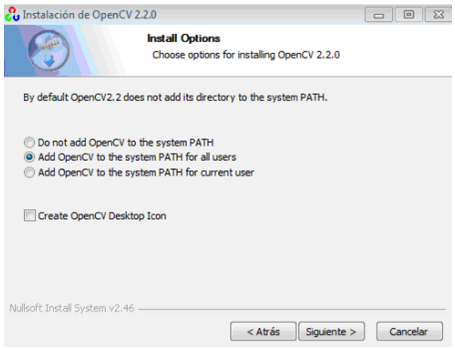

Figura A1: Instalación de las librerías OpenCV

3. *Descargar Visual C++ Express Edition*: se puede descargar gratuitamente desde la página web de Microsoft: <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express>

4. *Configurar Visual C++*: una vez descargado e instalado, lo ejecutamos y seleccionamos un nuevo proyecto de Win32 y Aplicación de consola Win32 y le damos a finalizar el asistente.

5. *Incluir librerías de OpenCV*: si queremos usarlas debemos incluirlas en el nuevo proyecto creado, para ello buscamos la pestaña "Propiedades" e incluimos los archivos "`C:\OpenCV2.2\include`", "`C:\OpenCV2.2\include\opencv`" y "`C:\OpenCV2.2\lib`" como se muestra en la figura A2.

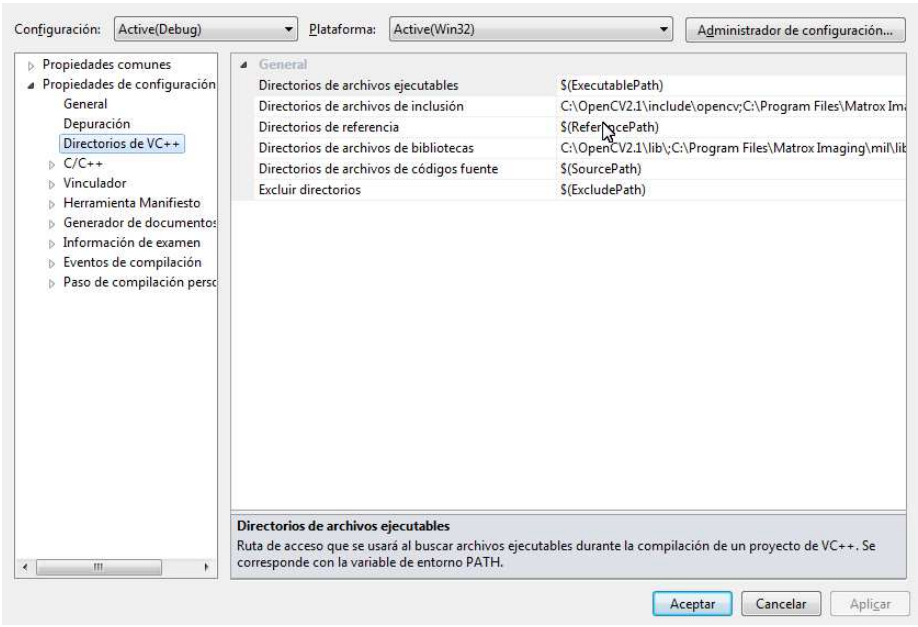

Figura A2: Archivos de inclusión y archivos de biblioteca

6. *Seleccionar las librerías de OpenCV*: por último hay que incluir las siguientes librerías en la sección que se muestra en la figura A3. opencv_calib3d220d.lib, opencv_contrib220d.lib, opencv_core220d.lib, opencv_features2d220d.lib, opencv_ffmpeg220d.lib, opencv_flann220d.lib, opencv_gpu220d.lib, opencv_highgui220d.lib, opencv_imgproc220d.lib, opencv_legacy220d.lib, opencv_ml220d.lib, opencv_objdetect220d.lib y opencv_video220d.lib

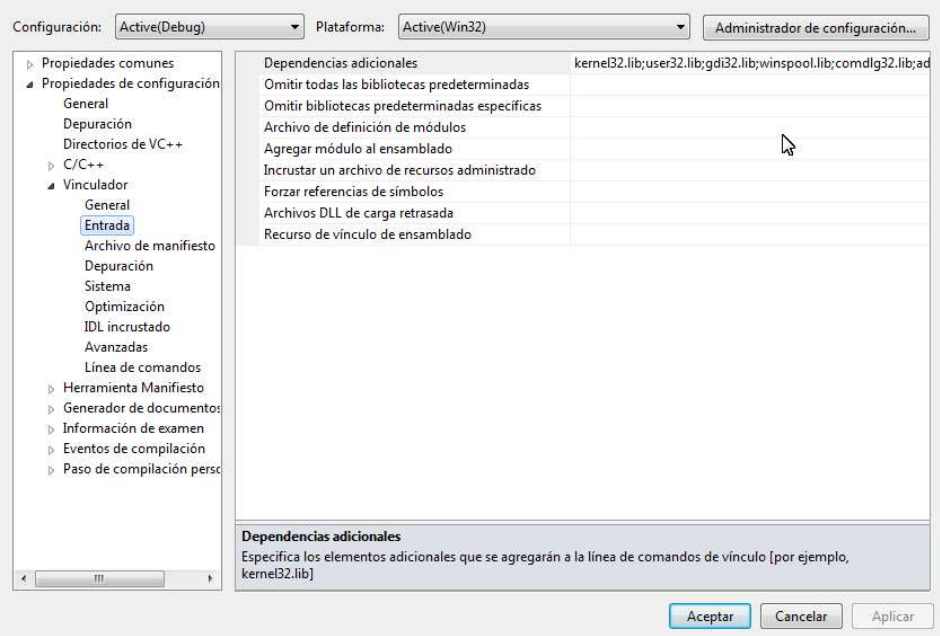

Figura A3: Librerías para el modo debug

Así pues, una vez seguidos estos seis pasos, ya estaría configurado nuestro proyecto y se podría empezar a programar la aplicación.

Antes de pasar a mostrar los códigos hay que destacar que ambos programas fueron modificados pues, finalmente, lo que reciben es la imagen estéreo ya dividida, es decir, reciben dos imágenes (derecha e izquierda) y no una como se explicó en el apartado 4.1.1, por tanto, ya no es necesario usar funciones como *cvSplit* de OpenCV para separarlas. También, al ser imágenes simples, pueden ser leídas en escala de grises y, en consecuencia, la etapa de conversión a escala de grises comentada en 5.1.1 es innecesaria.

Dicho esto, lo último que queda es presentar los códigos del proyecto, están divididos por aplicaciones (programa 1 y programa 2) y por funciones, y se ha intentado que el orden sea el de llamada a cada función para que resulte más intuitivo. Sin más dilación, en la siguiente página y sucesivas se presentan los códigos en lenguaje C++ que, a buen seguro, serán de interés para el lector.

```
//------------------------------------------------------------------------
// PROGRAMA 1
//------------------------------------------------------------------------
//            Main.cpp:            Main Program
//            Created by:          Luis Aranda Barjola
//            Start Date:          02/03/12
//            Modify Date:  05/08/12
//            University Carlos III de Madrid (Spain)
//------------------------------------------------------------------------
// PREPROCESSOR
//------------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

//------------------------------------------------------------------------
// DISPLAY ERROR FUNCTION
//------------------------------------------------------------------------
void DisplayErrorBox(LPTSTR lpszFunction);

//------------------------------------------------------------------------
// MAIN
//------------------------------------------------------------------------
int _tmain(int argc, TCHAR *argv[])
{
	//Variables Definition
	char *Imagename, *Imagename_p, *previmagename;
	int dir=0, error=0, pair=0, res=0, i=0, tama[70]={0}, simi[15]={0};
	float prevdist[100]={0};
	char p;
	WIN32_FIND_DATA ffd;
	LARGE_INTEGER filesize;
	TCHAR szDir[MAX_PATH];
	HANDLE hFind = INVALID_HANDLE_VALUE;
	DWORD dwError=0;

	//Reserve Memory
	previmagename = (char*)malloc(BUFFER_SIZE);

	//If Directory is not Specified as a Command-line Argument -> Print Usage
	if(argc != 2){
		_tprintf(TEXT("\nUsage: %s <directory name>\n"), argv[0]);
		return(-1);
	}

	//Display Chosen Directory
	_tprintf(TEXT("\nTarget directory is: %s\n\n"), argv[1]);

	//Prepare String for use with FindFile Functions
	//Copy the String to a Buffer, then Append '\*' to the Directory Name
	StringCchCopy(szDir, MAX_PATH, argv[1]);
	StringCchCat(szDir, MAX_PATH, TEXT("\\*"));

	//Find the First File in the Directory
	hFind = FindFirstFile(szDir, &ffd);

	if(INVALID_HANDLE_VALUE == hFind){
		DisplayErrorBox(TEXT("FindFirstFile"));
		return dwError;
	}
```

```
//List all the Files in the Directory with Some Info About Them
do{
	if(ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
		_tprintf(TEXT("  %s   <DIR>\n"), ffd.cFileName);
		dir=1;
	}
	else{
		filesize.LowPart = ffd.nFileSizeLow;
		filesize.HighPart = ffd.nFileSizeHigh;
		_tprintf(TEXT("  %s   %ld bytes\n"), ffd.cFileName,
		filesize.QuadPart);
	}
	//If the Item Detected isn't a Directory
	if(dir == 0){
		//Convert wchar* to char*
		Imagename = (char*)malloc(BUFFER_SIZE);
		wcstombs(Imagename, ffd.cFileName, BUFFER_SIZE);

		//Check that the Image hasn't been Processed
		for(i=0; i<BUFFER_SIZE; i++){
			p = Imagename[i];
			if(p == 'r'){
				//If processed -> End Program, No More Images
				if((Imagename[i+1] == 'i')&&(Imagename[i+2] ==
				'g')){
					printf("\n NO MORE IMAGES\n");
					printf(" ENDING PROGRAM...\n");
					exit(0);
				}
			}
		}

		//Even or Odd?
		pair++;
		res = pair % 2;

		//Only with Even
		if(res == 0){
			//Open Even Image
			error = OpenImage(Imagename);

			//Detect the Cars
			if(error == 0){
				CarsDetect(Imagename, Imagename_p, tama, simi,
				prevdist, previmagename);
			}
		}
		else{
			//Save Imagename
			Imagename_p = Imagename;
		}
	}
	getchar();
	dir=0;
}
while(FindNextFile(hFind, &ffd) != 0);

//Free Buffers
free(Imagename);
free(previmagename);
```

```
	dwError = GetLastError();
	if(dwError != ERROR_NO_MORE_FILES){
		DisplayErrorBox(TEXT("FindFirstFile"));
	}

	FindClose(hFind);
	return dwError;
}


//-------------------------------------------------------------------------
// DISPLAY ERROR FUNCTION
//-------------------------------------------------------------------------
void DisplayErrorBox(LPTSTR lpszFunction)
{
    //Retrieve the System Error Message for the Last-error Code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS, NULL, dw, MAKELANGID(LANG_NEUTRAL,
    SUBLANG_DEFAULT), (LPTSTR) &lpMsgBuf, 0, NULL);

    //Display the Error Message and Clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf, LocalSize(lpDisplayBuf) /
sizeof(TCHAR), TEXT("%s failed with error %d: %s"), lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

       //Free Buffers
    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

```
//------------------------------------------------------------------------
//            Main_Header.h:      Main Header with Preprocessor Instructions
//            Created by:         Luis Aranda Barjola
//            Start Date:         02/03/12
//            Modify Date:  05/08/12
//            University Carlos III de Madrid (Spain)
//------------------------------------------------------------------------
// INCLUDES
//------------------------------------------------------------------------
#include <cv.h>
#include <highgui.h>
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <tchar.h>
#include <stdlib.h>
#include <strsafe.h>
#include <ctype.h>
#include "cvaux.h"

using namespace cv;
using namespace std;
//------------------------------------------------------------------------
// FUNCTIONS HEADERS
//------------------------------------------------------------------------
#include "OpenImage.h"
#include "CarsDetect.h"
#include "DetectAndDraw.h"
#include "Edges.h"
#include "Histograms.h"
#include "Distance.h"
#include "Speed.h"

//------------------------------------------------------------------------
// DEFINES
//------------------------------------------------------------------------
#define BUFFER_SIZE 100
#define OSCILLATION 5
#define D_MAX 16
#define RESOLUCION 16
#define DISTMAX 60.0 //m
#define DISTMIN 3.0 //m
```

```
//---------------------------------------------------------------------
//            OpenImage.cpp:      Open an Image
//            Created by:         Luis Aranda Barjola
//            Start Date:         07/03/12
//            Modify Date:  25/06/12
//            University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

int OpenImage(char* ImageName)
{
	//Variables Definition
	IplImage *im;

	//Load Image and Check we Done Well
	im = cvLoadImage(ImageName, -1);
	if(im == 0){
		printf("  <Error Loading Image>\n");
		return 1;
	}

	//Open Image
	cvNamedWindow(ImageName, CV_WINDOW_AUTOSIZE);
	cvShowImage(ImageName, im);

	//Wait a Key
	cvWaitKey(0);
	cvReleaseImage(&im);
	cvDestroyWindow(ImageName);
	return 0;
}

//---------------------------------------------------------------------
//            CarsDetect.cpp:     Detect the Cars in the Image
//            Created by:         Luis Aranda Barjola
//            Start Date:         07/03/12
//            Modify Date:  05/08/12
//            University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

//Load Haar Cascade Name
String CascadeName = "C:/OpenCV2.2/Images/Cascades/haarcascade.xml";

int CarsDetect(char* ImageName, char* ImageName2, int* tama, int* simi, float*
prevdist, char* previmagename)
{
	//Variables Definition
	IplImage *right, *left, *ImaDisp;
	Mat imgR;
	int points[20]={0};
	float dist[100]={0};

	//Cascade Definition
	CascadeClassifier Cascade;

	//Load the Cascade
	Cascade.load(CascadeName);
```

```
	//Load 2 Images
	left = cvLoadImage(ImageName, CV_LOAD_IMAGE_GRAYSCALE);
	right = cvLoadImage(ImageName2, CV_LOAD_IMAGE_GRAYSCALE);

	//Transform IplImage to Mat
	imgR = cvCreateMat(right->height, right->width, CV_32FC1);
	imgR = Mat(right, true);

	//Call DetectAndDraw
	DetectAndDraw(imgR, Cascade, ImageName2, tama, simi, points);

	//Obtain Disparity Map
	ImaDisp = cvCreateImage(cvSize(left->width, left->height), IPL_DEPTH_16S,
	1);
	CvStereoBMState *StereoBMState = cvCreateStereoBMState(CV_STEREO_BM_BASIC,
	D_MAX);
	StereoBMState->SADWindowSize = 21;
	cvFindStereoCorrespondenceBM(left, right, ImaDisp, StereoBMState);
	cvSmooth(ImaDisp, ImaDisp, CV_MEDIAN, 3, 3, 0, 0);
	cvNormalize(ImaDisp, ImaDisp, 0, 255, CV_MINMAX);
	cvSaveImage("Map.jpg", ImaDisp);

	//Calculate Distances
	if(points[0] != 0){
		Distance(ImaDisp, points, tama[69], dist);
	}

	//Calculate Speed
	Speed(ImageName2, previmagename, dist, prevdist);

	//Release Images
	cvReleaseImage(&right);
	cvReleaseImage(&left);
	return 0;
}

//---------------------------------------------------------------------
//		DetectAndDraw.cpp:  Draw Rectangles & Lines in the Image
//		Created by:			Luis Aranda Barjola
//		Start Date:			08/03/12
//		Modify Date:		26/07/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void DetectAndDraw(Mat& img, CascadeClassifier& Cascade, char* ImageName, int*
tama, int* simi, int* points)
{
	//Variables Definition
	int i=0, j=0, k=0, k2=0, m=0, q=0, media=0, up, down, rows, area, sim,
	option[8]={0};
	int eq=0, big=0, sma=0, x1=0, y1=0, y2=0, err=0, ft=1, dif=0, dif2=0,
	aux=0, aux2=0, aux3=0;
	char p;
	char *name;
	vector<Rect> cars;
	Point vert1, vert2;
```

```
//Definition of Colors Just in Case There are Many Cars
const static Scalar colors[] = {CV_RGB(0,0,255)};
Scalar color = colors[0];

//Create a Gray and Smaller Image
Mat gray, smallImg(cvRound(img.rows*2/3), cvRound(img.cols*8/10),
CV_8UC1);

//Compares the Image with the Cascade
Cascade.detectMultiScale(img, cars, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE,
Size(40, 40));

//Plot Rectangles for Every Car Detected
for(vector<Rect>::const_iterator r = cars.begin(); r != cars.end(); r++,
i++){
	//Average of Pixels Under the Image Rectangle (Last Filter)
	media=0;
	for(j=0; j<=r->width*img.step/img.cols*8/10; j++){
		media = media + img.data[(r->y + r->height)*img.step+(r-
		>x)*img.step/img.cols*8/10+j];
	}
	media = media / (j-1);
	if((media < 160)||(tama[69] != 0)){

		//Rectangle
		vert1.x = r->x;
		vert1.y = r->y;
		vert2.x = (r->x + r->width);
		vert2.y = (r->y + r->height);
		//--------------------------------
		// Horizontal Edges Detection
		//--------------------------------
		//Load Original Image
		IplImage *img1 = cvLoadImage(ImageName,
		CV_LOAD_IMAGE_GRAYSCALE);

		//Create ROI
		cvSetImageROI(img1, cvRect(vert1.x, vert1.y, vert2.x-vert1.x,
		vert2.y-vert1.y));
		IplImage *img2 = cvCreateImage(cvGetSize(img1), img1->depth,
		img1->nChannels);

		//Copy Subimage
		cvCopy(img1, img2, NULL);

		//Call Horizontal Edges Detector
		Edges(img2, up, down, rows);

		//Up & Down Coordinates
		x1 = vert1.x + (vert2.x-vert1.x)/2;
		y1 = vert1.y + up;
		y2 = vert1.y + down;
```

```
//Compare Areas
area = rows;
for(m=0; m<7; m++){
	if(tama[m] == area){
		eq++;
	}
	if((tama[m] < area)&&(tama[m] != 0)){
		big++;
	}
	if(tama[m] > area){
		sma++;
	}
}
//----------------------------
// New Car(s) Appear(s)
//----------------------------
//Check if it's a New Car or Detected Before
for(m=0; m<7; m++){
	if(tama[m] != 0){
		if((abs(tama[8+3*m]-x1)) <= 40){
			//Detected Before
			ft=0;
		}
	}
	else{break;}
}
//New Car
if(ft == 1){
	//Save New Information
	tama[k+33] = rows;
	tama[41+3*k] = x1;
	tama[42+3*k] = y1;
	tama[43+3*k] = y2;
	ft=1;
}
//----------------------------
// Car(s) Moving
//----------------------------
else{
	//-------
	// CASES
	//-------
	//"Nothing Has Changed" or "Same Car Moved" or "New Car" or "Error"
	if(eq >= 1){
		//Nothing Has Changed
		if((x1 == tama[8+3*m])&&(y1 == tama[9+3*m])&&(y2 == tama[10+3*m])){
			option[k] = 1;
		}
		else{
			for(m=0; m<7; m++){
				//Same Car Moved
				if((abs(tama[8+3*m]-x1)) <= 5){
					option[k] = 2;
					//Upper Point Oscillation
					do{
						y1 = cvRound((y1+tama[9+3*m])/2);
					}while((abs(tama[9+3*m]-y1)) > OSCILLATION);
```

```
                                        //Lower Point Oscillation
                                        do{
                                                y2 = cvRound((y2+tama[10+3*m])/2);
                                        }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                        break;
                                }
                                else{
                                        //Error
                                        if(m == 6){err=1;}
                                }
                        }
                }
        }
//Common Cases
if(eq == 0){
        //Biggest Car
        if(sma == 0){
                for(m=0; m<7; m++){
                        if(((area-tama[m]) <= 5)||((abs(tama[8+3*m]-x1)) <= 40)){
                                if((abs(tama[8+3*m]-x1)) <= 40){
                                        option[k] = 3;
                                        //Upper Point Oscillation
                                        do{
                                                y1 = cvRound((y1+tama[9+3*m])/2);
                                        }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                                        //Lower Point Oscillation
                                        do{
                                                y2 = cvRound((y2+tama[10+3*m])/2);
                                        }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                        break;
                                }
                                //First Time
                                else{ft=1;}
                        }
                }
        }

        //Smallest Car
        if(big == 0){
                for(m=0; m<7; m++){
                        if(((tama[m]-area) <= 5)||((abs(tama[8+3*m]-x1)) <= 25)){
                                if((abs(tama[8+3*m]-x1)) <= 25){
                                        option[k] = 4;
                                        //Upper Point Oscillation
                                        do{
                                                y1 = cvRound((y1+tama[9+3*m])/2);
                                        }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                                        //Lower Point Oscillation
                                        do{
                                                y2 = cvRound((y2+tama[10+3*m])/2);
                                        }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                        break;
                                }
                                //First Time
                                else{ft=1;}
                        }
                }
        }
```

```
	//"Biggest Car Moves Away" or "Other Car Approach" or "Biggest Car Hid a
Car" or "Error"
	if((big > sma)&&(sma != 0)){
		for(m=0; m<7; m++){
			if((((area-tama[m]) <= 5)&&((area-tama[m]) >
			0))||((abs(tama[8+3*m]-x1)) <= 15)){
				//Other Car Approach
				if((abs(tama[9+3*m]-
				tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
					option[k] = 5;
					//Upper Point Oscillation
					do{
						y1 = cvRound((y1+tama[9+3*m])/2);
					}while((abs(tama[9+3*m]-y1)) > OSCILLATION);
					//Lower Point Oscillation
					do{
						y2 = cvRound((y2+tama[10+3*m])/2);
					}while((abs(tama[10+3*m]-y2)) > OSCILLATION);
					break;
				}
				//Error
				if(m == 6){err=1;}
			}
			if((((tama[m]-area) <= 5)&&((tama[m]-area) >
			0))||((abs(tama[8+3*m]-x1)) <= 15)){
				//Biggest Car Moves Away
				if((abs(tama[9+3*m]-
				tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
					option[k] = 6;
					//Upper Point Oscillation
					do{
						y1 = cvRound((y1+tama[9+3*m])/2);
					}while((abs(tama[9+3*m]-y1)) > OSCILLATION);
					//Lower Point Oscillation
					do{
						y2 = cvRound((y2+tama[10+3*m])/2);
					}while((abs(tama[10+3*m]-y2)) > OSCILLATION);
					break;
				}
				//Error
				if(m == 6){err=1;}
			}
			//Biggest Car Hid a Car
			else{
				if(m == 6){err=2;}
			}
		}
	}
	//"Smallest Car Approach" or "Other Car Moves Away" or "Other Cars Hid a
Car" or "Error"
	if((big < sma)&&(big != 0)){
		for(m=0; m<7; m++){
			if((((area-tama[m]) <= 5)&&((area-tama[m]) >
			0))||((abs(tama[8+3*m]-x1)) <= 15)){
				//Smallest Car Approach
				if((abs(tama[9+3*m]-
				tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
					option[k] = 7;
```

```
                                //Upper Point Oscillation
                                do{
                                        y1 = cvRound((y1+tama[9+3*m])/2);
                                }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                                //Lower Point Oscillation
                                do{
                                        y2 = cvRound((y2+tama[10+3*m])/2);
                                }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                break;
                        }
                        //Error
                        if(m == 6){err=1;}
                }
                if((((tama[m]-area) <= 5)&&((tama[m]-area) >
                0))||((abs(tama[8+3*m]-x1)) <= 15)){
                        //Other Car Moves Away
                        if((abs(tama[9+3*m]-
                        tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
                                option[k] = 8;
                                //Upper Point Oscillation
                                do{
                                        y1 = cvRound((y1+tama[9+3*m])/2);
                                }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                                //Lower Point Oscillation
                                do{
                                        y2 = cvRound((y2+tama[10+3*m])/2);
                                }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                break;
                        }
                        //Error
                        if(m == 6){err=1;}
                }
                //Other Cars Hid a Car
                else{
                        if(m == 6){err=2;}
                }
        }
}

//"Car Approach" or "Car Moves Away" or "Other Cars Hid a Car" or "Error"
if(big == sma){
        for(m=0; m<7; m++){
                if((((area-tama[m]) <= 5)&&((area-tama[m]) >
                0))||((abs(tama[8+3*m]-x1)) <= 15)){
                        //Car Approach
                        if((abs(tama[9+3*m]-
                        tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
                                option[k] = 9;
                                //Upper Point Oscillation
                                do{
                                        y1 = cvRound((y1+tama[9+3*m])/2);
                                }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                                //Lower Point Oscillation
                                do{
                                        y2 = cvRound((y2+tama[10+3*m])/2);
                                }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                                break;
                        }
```

```
                        //Error
                        if(m == 6){err=1;}
                    }
                    if((((tama[m]-area) <= 5)&&((tama[m]-area) >
                    0))||((abs(tama[8+3*m]-x1)) <= 15)){
                        //Car Moves Away
                        if((abs(tama[9+3*m]-
                        tama[m])<y1)&&((tama[10+3*m]+tama[m])>y2)){
                            option[k] = 10;
                            //Upper Point Oscillation
                            do{
                                y1 = cvRound((y1+tama[9+3*m])/2);
                            }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                            //Lower Point Oscillation
                            do{
                                y2 = cvRound((y2+tama[10+3*m])/2);
                            }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                            break;
                        }
                        //Error
                        if(m == 6){err=1;}
                    }
                    //Other Cars Hid a Car
                    else{
                        if(m == 6){err=2;}
                    }
                }
            }
        }
        //Special Case, Hidden Car
        if(err == 2){
            //Calculate Center of the Rectangle
            int cy = cvRound((y1+y2)/2);
            for(m=0; m<7; m++){
                int vr = tama[8+3*m] + cvRound(tama[m]/2);
                int vl = tama[8+3*m] - cvRound(tama[m]/2);
                int hd = tama[10+3*m] + cvRound(tama[m]/4);
                int hu = tama[9+3*m] - cvRound(tama[m]/2);
                //Hidden Car
                if(((x1 > vl)&&(x1 < vr))&&((cy > hu)&&(cy < hd))){
                    option[k] = 11;
                    //Upper Point Oscillation
                    do{
                        y1 = cvRound((y1+tama[9+3*m])/2);
                    }while((abs(tama[9+3*m]-y1)) > OSCILLATION);
                    //Lower Point Oscillation
                    do{
                        y2 = cvRound((y2+tama[10+3*m])/2);
                    }while((abs(tama[10+3*m]-y2)) > OSCILLATION);
                    err=0;
                    break;
                }
                //Error
                else{err=1;}
            }
        }
    }
```

```
//If OK We Draw Vertical Line
if(err == 0){
//-------------------------------------
// Vertical Line Histogram (Searching)
//-------------------------------------
//Load Original Image
img1 = cvLoadImage(ImageName, CV_LOAD_IMAGE_GRAYSCALE);

//One Time For New Cars
if(ft == 1){
	//Draw Line & Rectangle
	CvPoint p1 = cvPoint(x1, y1);
	CvPoint p2 = cvPoint(x1, y2);
	line(img, p1, p2, color, 2, 8, 0);
	rectangle(img, vert1, vert2, color, 3, 8, 0);

	//Create ROI
	cvSetImageROI(img1, cvRect(x1-3, y1, 6, y2-y1));
	img2 = cvCreateImage(cvGetSize(img1), img1->depth, img1-
	>nChannels);

	//Copy Subimage
	cvCopy(img1, img2, NULL);

	//Call Histograms
	Histograms(img2, sim);

	//Free ROI & Images
	cvResetImageROI(img1);
	cvReleaseImage(&img2);
	cvReleaseImage(&img1);
}
else{
	//Search Car
	for(m=0; m<7; m++){
		if(tama[m] != 0){
			if(abs(tama[8+3*m]-x1) <= 15){
				dif = abs(tama[8+3*m]-x1);
				if((dif < aux)||(aux == 0)||(dif == 0)){
					aux = dif;
					q = m;
				}
			}
		}
	else{break;}
}
dif = aux;
aux = 0;

//Search More Similar Histogram
for(m=0; m<=2*dif; m++){
	//Create ROI
	cvSetImageROI(img1, cvRect(x1-dif+m, y1, 6, tama[10+3*q]-
	tama[9+3*q]));

	img2 = cvCreateImage(cvGetSize(img1), img1->depth, img1-
	>nChannels);

	//Copy Subimage
	cvCopy(img1, img2, NULL);
```

```
                //Call Histograms
                Histograms(img2, sim);

                //Save More Similar
                dif2 = abs(simi[q]-sim);
                if((dif2 < aux)||(aux == 0)||(dif2 == 0)){
                        if(dif2 == 0){
                                aux2 = sim;
                                aux++;
                        }
                        else{
                                aux = dif2;
                                aux2 = sim;
                                aux3 = x1-dif+m;
                        }
                }
                //Release Image
                cvReleaseImage(&img2);
        }
        //Line & Rectangle Relocation
        sim = aux2;
        x1 = aux3+3;
        vert1.x = x1-cvRound(rows/2);
        vert2.x = x1+cvRound(rows/2);

        //Draw Line & Rectangle
        CvPoint p1 = cvPoint(x1, y1);
        CvPoint p2 = cvPoint(x1, y2);
        line(img, p1, p2, color, 2, 8, 0);
        rectangle(img, vert1, vert2, color, 3, 8, 0);

        //Free ROI & Image
        cvResetImageROI(img1);
        cvReleaseImage(&img1);
}
        //------------------------------
        // Save Information
        //------------------------------
        //Save the Rectangle Area, Upper & Lower Points in Tama
        tama[k+33] = rows;
        tama[41+3*k] = x1;
        tama[42+3*k] = y1;
        tama[43+3*k] = y2;

        //Save Medium Vertical Line Point
        points[k2] = x1;
        points[k2+1] = cvRound((y1+y2)/2);
        k2 = k2 + 2;

        //Save the Similarity in Simi
        simi[k+8] = sim;
        k++;
}
        //If not OK Do Nothing
        if(err == 1){
                printf("Error\n");
        }
```

```
        //Reset Counters
        eq=0; big=0; sma=0; err=0; ft=1; dif=0; dif2=0; aux=0; aux2=0; aux3=0;
        }
}

        //Delete Old Information & Copy New
        for(i=0; i<7; i++){
                tama[i] = 0;
                tama[8+3*i] = 0;
                tama[9+3*i] = 0;
                tama[10+3*i] = 0;
                simi[i] = 0;
        }
        for(i=0; i<k; i++){
                tama[i] = tama[i+33];
                tama[8+3*i] = tama[41+3*i];
                tama[9+3*i] = tama[42+3*i];
                tama[10+3*i] = tama[43+3*i];
                simi[i] = simi[i+8];
        }
        //Save Number of Cars in the Image
        tama[69] = k;

        //Image Rename Loop
        name = ImageName;
        for(i=0; i<BUFFER_SIZE; i++){
                p = name[i];
                if(p == '.'){
                        name[i] = '_';
                        name[i+1] = 'r';
                        name[i+2] = 'i';
                        name[i+3] = 'g';
                        strcat(name, ".tiff");
                        break;
                }
        }
        //Write a New Image
        imwrite(name, img);

        //Open the New Image
        OpenImage(name);
}
```

```
//---------------------------------------------------------------------
//            Edges.cpp:          Determine Upper & Lower Edges of a Car
//            Created by:         Luis Aranda Barjola
//            Start Date:         23/03/12
//            Modify Date:  26/07/12
//            University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

int Edges(IplImage* imROI, int &pup, int &pdown, int &row)
{
	//Variables Definition
	IplImage *im_gray, *im_sob;
	int col, x, y, temp=0, vect[3000]={0};

	//-----------------------------
	// Horizontal Edge Detection
	//-----------------------------
	//Create New Grayscale Image
	CvSize size_im = cvSize(imROI->width, imROI->height);
	im_gray = cvCreateImage(size_im, imROI->depth, 1);

	//Smooth Image with a Gaussian
	cvSmooth(imROI, im_gray, CV_GAUSSIAN, 5);

	//Equalize Image
	cvEqualizeHist(im_gray, im_gray);

	//Detect Horizontal Edges using Sobel
	im_sob = cvCreateImage(cvGetSize(im_gray), IPL_DEPTH_32F, 1);
	cvSobel(im_gray, im_sob, 0, 1, 3);

	//Threshold Image
	cvThreshold(im_sob, im_sob, 100, 255, CV_THRESH_BINARY);

	//-----------------------------
	// Upper & Lower Points
	//-----------------------------
	//Transform IplImage to Mat
	CvMat *mat = cvCreateMat(im_sob->height, im_sob->width, CV_8UC1);
	cvConvert(im_sob, mat);

	//Detemine Image Number of Rows and Columns
	for(row=0; row < mat->rows; row++);
	for(col=0; col < mat->cols; col++);

	//Get Number of White Pixels in Every Row
	for(x=0; x<row; x++){
		for(y=0; y<col; y++){
			temp = temp + CV_MAT_ELEM(*mat, int, x, y);
		}
		//Save the Number of White Pixels in One Row
		vect[x] = cvRound(abs(temp)/255);
		temp=0;
	}
```

```
	//Search Upper Edge
	for(x=0; x < ((2*row)/5); x++){
		if(temp < vect[x]){
			//Save Biggest Value in "pup"
			pup = x;
			temp = vect[x];
		}
	}
	temp=0;

	//Search Lower Edge
	for(x=((3*row)/5); x < (row-5); x++){
		if(temp < vect[x]){
			//Save Biggest Value in "pdown"
			pdown = x;
			temp = vect[x];
		}
	}
	cvReleaseImage(&im_gray);
	cvReleaseImage(&im_sob);
	return 0;
}

//---------------------------------------------------------------------
//		Histograms.cpp:		Determine Histogram of Vertical Line
//		Created by:		Luis Aranda Barjola
//		Start Date:		24/04/12
//		Modify Date:	26/07/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Histograms(IplImage* imROI, int &IR)
{
	//Variables Definition
	IplImage *im_gray;
	int row, col, x, y, temp=0, i=0, j=0, aux=0, vect[3000]={0};

	//---------------------------
	// Horizontal Edge Detection
	//---------------------------
	//Create New Grayscale Image
	CvSize size_im = cvSize(imROI->width, imROI->height);
	im_gray = cvCreateImage(size_im, imROI->depth, 1);

	//Smooth Image with a Gaussian
	cvSmooth(imROI, im_gray, CV_GAUSSIAN, 5);

	//Equalize Image
	cvEqualizeHist(im_gray, im_gray);

	//Transform IplImage to Mat
	CvMat *mat = cvCreateMat(im_gray->height, im_gray->width, CV_8UC1);
	cvConvert(im_gray, mat);

	//Detemine Image Number of Rows and Columns
	for(row=0; row < mat->rows; row++);
	for(col=0; col < mat->cols; col++);
```

```
	//Get Pixels Value in Every Row
	for(x=0; x<row; x++){
		for(y=0; y<col; y++){
			temp = temp + CV_MAT_ELEM(*mat, int, x, y);
		}
		//Save Pixels Numeric Value in One Row
		vect[x] = abs(temp);
		temp=0;
	}

	//-----------------------------
	// Interquartile Range
	//-----------------------------
	//Sort Vector Lowest to Highest
	for(i=1; i<row; i++){
		for(j=0; j<row-i; j++){
			if(vect[j] > vect[j+1]){
				aux = vect[j];
				vect[j] = vect[j+1];
				vect[j+1] = aux;
			}
		}
	}

	//First Quartile
	i = cvRound(row/4);

	//Third Quartile
	j = cvRound((3*row)/4);

	//Interquartile Range
	IR = vect[j] - vect[i];

	//Make it Proportional to the Area
	i = (imROI->width)*(imROI->height);
	IR = cvRound(IR/i);

	//Release Image
	cvReleaseImage(&im_gray);
}

//---------------------------------------------------------------------
//		Distance.cpp: Calculate Distances in the Image
//		Created by:		Luis Aranda Barjola
//		Start Date:		28/07/12
//		Modify Date:	05/08/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

//Stereo Camera Parameters
struct StereoCamera{
	float Baseline; //m
	float FocalLength;
	float CenterRow;
	float CenterCol;
};
```

```
int Distance(IplImage* DispMap, int* points, int ncars, float* dist)
{
	//Variables Definition
	short val;
	short *porig16;
	unsigned char *p_imadisp;
	float u, v, X, Y, Z, D, f, fval;
	double DispMin, DispMax;
	int i=0, j=1, k=0;

	//Camera Definition
	struct StereoCamera Camera;
	Camera.Baseline = 0.119915; //m
	Camera.FocalLength = 811.9104;
	Camera.CenterRow = 247.1637;
	Camera.CenterCol = 321.561;
	//Calculate Min & Max Disparity
	cvMinMaxLoc(DispMap, &DispMin, &DispMax);
	//Camera Parameters
	f = Camera.FocalLength;
	D = Camera.Baseline;
	//Disparity
	p_imadisp = (unsigned char*)DispMap->imageData;

	//Obtain Distance for Every Car
	for(i=0; i<ncars; i++){
		porig16 = (short*)(p_imadisp + points[2*i+1]*DispMap->widthStep);
		porig16 += points[2*i];
		val =* porig16;
		if((val > DispMin) && (val < DispMax)){
			fval = val/RESOLUCION;
			Z = f*D/fval;
			if(Z <= DISTMAX){
				u = points[2*i] - Camera.CenterCol;
				v = points[2*i+1] - Camera.CenterRow;
				X = Z*u/f;
				Y = Z*v/f;
				dist[k] = Z;
				dist[k+1] = X;
				dist[k+2] = Y;
				k = k + 3;
			}
		}
	}

	//Show Results
	if(dist[0] != 0){
		printf(" -----------------\n");
		printf(" |   DISTANCES   |\n");
		printf(" -----------------\n");
		i=0; j=1;
		while(dist[i] != 0){
			printf(" ----- CAR %d -----\n", j);
			printf(" | X: %fm |\n", dist[i+1]);
			printf(" | Y: %fm |\n", dist[i+2]);
			printf(" | Z: %fm |\n", dist[i]);
			i = i + 3;
			j++;
		}
		printf(" -----------------\n");
```

```
	}
	else{
		printf(" --- NO RESULTS ---\n");
	}

	return 0;
}

//---------------------------------------------------------------------
//		Speed.cpp:		Calculate Cars Speed
//		Created by:		Luis Aranda Barjola
//		Start Date:		04/08/12
//		Modify Date:	05/08/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Speed(char* Imagename, char* previmagename, float* dist, float* prevdist)
{
	//Variables Definition
	int i=0, j=0, k=0, more=0, t1=0, t2=0, Dt=0, t1aux=0, t2aux=0, t3aux=0;
	float v=0, Ddist[20]={0};
	char p1, p2;

	//First Time
	if((prevdist[0] == 0)||(dist[0] == 0)){
		//Save Information
		for(i=0; i<BUFFER_SIZE; i++){
			previmagename[i] = Imagename[i];
		}
		for(i=0; i<100; i++){
			prevdist[i] = dist[i];
		}
	}
	//Following Times
	else{
		//----------
		// Distance
		//----------
		//More or Less Cars than Before?
		for(k=0; k<32; k++){
			if((dist[3*k] == 0)&&(prevdist[3*k] == 0)){
				break;
			}
			else{
				//More Distances than Before -> New Cars
				if((dist[3*k] != 0)&&(prevdist[3*k] == 0)){
					more = 1;
				}
			}
		}
		if(more == 1){
			//More New Cars
			while(prevdist[i] != 0){
				//Calculate Distance Increment
				Ddist[j] = abs(dist[i]-prevdist[i]);
				//Increment Counters
				i = i + 3;
				j++;
```

```
		}
	}
	else{
		//Less or Equal Cars
		while(dist[i] != 0){
			//Calculate Distance Increment
			Ddist[j] = abs(dist[i]-prevdist[i]);

			//Increment Counters
			i = i + 3;
			j++;
		}
	}
	//------
	// Time
	//------
	//Calculate T1
	for(i=0; i<BUFFER_SIZE; i++){
		p1 = Imagename[i];
		if(p1 == 's'){
			//Hundreds
			p1 = Imagename[i+4];
			t1aux = p1 - '0';
			//Tens
			p1 = Imagename[i+5];
			t2aux = p1 - '0';
			//Units
			p1 = Imagename[i+6];
			if(p1 == '.'){
				t1 = t1aux*10 + t2aux;
			}
			else{
				t3aux = p1 - '0';
				t1 = t1aux*100 + t2aux*10 + t3aux;
			}
			break;
		}
	}
	//Calculate T2
	for(i=0; i<BUFFER_SIZE; i++){
		p2 = previmagename[i];
		if(p2 == 's'){
			//Hundreds
			p2 = previmagename[i+4];
			t1aux = p2 - '0';
			//Tens
			p2 = previmagename[i+5];
			t2aux = p2 - '0';
			//Units
			p2 = previmagename[i+6];
			if(p2 == '.'){
				t2 = t1aux*10 + t2aux;
			}
			else{
				t3aux = p2 - '0';
				t2 = t1aux*100 + t2aux*10 + t3aux;
			}
			break;
		}
	}
```

```
                //Calculate Time Increment
                Dt = t1 - t2;
                if(Dt < 0){
                        Dt = 1000 - abs(Dt);
                }

                //-------
                // Speed
                //-------
                //Reset Counter
                i=0;

                //Calculate & Show Speed
                printf(" -----------------\n");
                printf(" |  SPEED (Km/h) |\n");
                printf(" -----------------\n");
                for(i=0; i<20; i++){
                        if(Ddist[i] != 0){
                                v = Ddist[i]/Dt;
                                v = v*3600; //Km/h
                                printf(" | Car %d: %3.2f |\n", i+1, v);
                        }
                        else{
                                if(more == 1){
                                        if(prevdist[3*i] != 0){
                                                printf(" | Car %d: V->Cte |\n", i+1);
                                        }
                                        else{
                                                break;
                                        }
                                }
                                else{
                                        if(dist[3*i] != 0){
                                                printf(" | Car %d: V->Cte |\n", i+1);
                                        }
                                        else{
                                                break;
                                        }
                                }
                        }
                }
                printf(" -----------------\n");

                //Update Values
                for(i=0; i<BUFFER_SIZE; i++){
                        previmagename[i] = Imagename[i];
                }
                for(i=0; i<100; i++){
                        prevdist[i] = dist[i];
                }
        }
}
```

```
//---------------------------------------------------------------------
// PROGRAMA 2
//---------------------------------------------------------------------
//            Main.cpp:            Main program
//            Created by:          Luis Aranda Barjola
//            Start Date:          02/03/12
//            Modify Date:  06/08/12
//            University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
// PREPROCESSOR
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

//---------------------------------------------------------------------
// DISPLAY ERROR FUNCTION
//---------------------------------------------------------------------
void DisplayErrorBox(LPTSTR lpszFunction);

//---------------------------------------------------------------------
// MAIN
//---------------------------------------------------------------------
int _tmain(int argc, TCHAR *argv[])
{
	//Variables Definition
	char* Imagename, *Imagename_p, *previmagename;
	int dir=0, error=0, pair=0, res=0, i=0, simi[15]={0}, coord[60]={0};
	float prevdist[100]={0};
	char p;
	WIN32_FIND_DATA ffd;
	LARGE_INTEGER filesize;
	TCHAR szDir[MAX_PATH];
	HANDLE hFind = INVALID_HANDLE_VALUE;
	DWORD dwError=0;

	//Reserve Memory
	previmagename = (char*)malloc(BUFFER_SIZE);

	//If Directory is not Specified as a Command-line Argument, Print Usage.
	if(argc != 2){
		_tprintf(TEXT("\nUsage: %s <directory name>\n"), argv[0]);
		return (-1);
	}

	//Display Chosen Directory
	_tprintf(TEXT("\nTarget directory is: %s\n\n"), argv[1]);

	//Prepare String for use with FindFile Functions
	//Copy the String to a Buffer, then Append '\*' to the Directory Name
	StringCchCopy(szDir, MAX_PATH, argv[1]);
	StringCchCat(szDir, MAX_PATH, TEXT("\\*"));

	//Find the First File in the Directory
	hFind = FindFirstFile(szDir, &ffd);

	if(INVALID_HANDLE_VALUE == hFind){
		DisplayErrorBox(TEXT("FindFirstFile"));
		return dwError;
	}
```

```
//List all the Files in the Directory with Some Info About Them
do{
	if(ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
		_tprintf(TEXT("  %s   <DIR>\n"), ffd.cFileName);
		dir=1;
	}
	else{
		filesize.LowPart = ffd.nFileSizeLow;
		filesize.HighPart = ffd.nFileSizeHigh;
		_tprintf(TEXT("  %s   %ld bytes\n"), ffd.cFileName,
		filesize.QuadPart);
	}
	//If the Item Detected isn't a Directory
	if(dir == 0){
		//Convert wchar* to char*
		Imagename = (char*)malloc(BUFFER_SIZE);
		wcstombs(Imagename, ffd.cFileName, BUFFER_SIZE);

		//Check that the Image hasn't been Processed
		for(i=0; i<BUFFER_SIZE; i++){
			p = Imagename[i];
			if(p == 'r'){
				//If processed -> End Program, No More Images
				if((Imagename[i+1] == 'i')&&(Imagename[i+2] ==
				'g')){
					printf("\n NO MORE IMAGES\n");
					printf(" ENDING PROGRAM...\n");
					exit(0);
				}
			}
		}

		//Even or Odd?
		pair++;
		res = pair % 2;

		//Only with Even
		if(res == 0){
			//Open Even Image
			error = OpenImage(Imagename);

			//Detect the Cars
			if(error == 0){
				CarsDetect(Imagename, Imagename_p, simi, coord,
				prevdist, previmagename);
			}
		}
		else{
			//Save Imagename
			Imagename_p = Imagename;
		}
	}
	getchar();
	dir=0;
}
while (FindNextFile(hFind, &ffd) != 0);

//Free Buffer
free(Imagename);
free(previmagename);
```

```
	dwError = GetLastError();
	if (dwError != ERROR_NO_MORE_FILES){
		DisplayErrorBox(TEXT("FindFirstFile"));
	}

	FindClose(hFind);
	return dwError;
}

//-------------------------------------------------------------------------
// DISPLAY ERROR FUNCTION
//-------------------------------------------------------------------------
void DisplayErrorBox(LPTSTR lpszFunction)
{
    //Retrieve the System Error Message for the Last-error Code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS, NULL, dw, MAKELANGID(LANG_NEUTRAL,
SUBLANG_DEFAULT), (LPTSTR) &lpMsgBuf, 0, NULL);

    //Display the Error Message and Clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf, LocalSize(lpDisplayBuf) /
sizeof(TCHAR), TEXT("%s failed with error %d: %s"), lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

       //Free Buffers
    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}

//-------------------------------------------------------------------------
//            CarsDetect.cpp:     Detect the Cars in the Image
//            Created by:         Luis Aranda Barjola
//            Start Date:         07/03/12
//            Modify Date:  06/08/12
//            University Carlos III de Madrid (Spain)
//-------------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

int CarsDetect(char* ImageName, char* ImageName2, int* simi, int* coord, float*
prevdist, char* previmagename)
{
	//Variables Definition
	IplImage *right, *left, *ImaDisp;
	Mat imgR;
	int i=0, points[1000]={0}, option[7]={0};
	float dist[100]={0};
	char *name;
	char p;

	//Load 2 Images
	left = cvLoadImage(ImageName, CV_LOAD_IMAGE_GRAYSCALE);
	right = cvLoadImage(ImageName2, CV_LOAD_IMAGE_GRAYSCALE);
```

```
//Transform IplImage to Mat
imgR = cvCreateMat(right->height, right->width, CV_32FC1);
imgR = Mat(right, true);

//Call Parts
Parts(imgR, points, option);

//Obtain Disparity Map
ImaDisp = cvCreateImage(cvSize(left->width,left->height), IPL_DEPTH_16S,
1);
CvStereoBMState *StereoBMState = cvCreateStereoBMState(CV_STEREO_BM_BASIC,
D_MAX);
StereoBMState->SADWindowSize = 21;
cvFindStereoCorrespondenceBM(left, right, ImaDisp, StereoBMState);
cvSmooth(ImaDisp, ImaDisp, CV_MEDIAN, 3, 3, 0, 0);
cvNormalize(ImaDisp, ImaDisp, 0, 255, CV_MINMAX);
cvSaveImage("Map.jpg", ImaDisp);

//Calculate Distances
if(points[0] != 0){
	Distance(imgR, ImaDisp, points, option, dist);
}

//When Detect a Car -> Searching
if(dist[0] != 0){
	Searching(imgR, ImageName2, simi, coord, points, option);
}
else{
	printf("\n NO CAR DETECTED\n");
	//Reset Simi & Coord
	for(i=0; i<15; i++){
		simi[i] = 0;
	}
	for(i=0; i<60; i++){
		coord[i] = 0;
	}
}

//Calculate Speed
Speed(ImageName2, previmagename, dist, prevdist);

//Image Rename Loop
name = ImageName2;
for(i=0; i<BUFFER_SIZE; i++){
	p = name[i];
	if(p == '.'){
		name[i] = '_';
		name[i+1] = 'r';
		name[i+2] = 'i';
		name[i+3] = 'g';
		strcat(name, ".tiff");
		break;
	}
}
//Write a New Image
imwrite(name, imgR);

//Open the New Image
OpenImage(name);
```

```
	//Release Images
	cvReleaseImage(&right);
	cvReleaseImage(&left);
	return 0;
}

//---------------------------------------------------------------------
//		Parts.cpp:		Detect Parts of the Car & Group Them
//		Created by:		Luis Aranda Barjola
//		Start Date:		23/06/12
//		Modify Date:	06/08/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Parts(Mat& img, int* points, int* option)
{
	//Variables Definition
	int corner[160]={0}, twocorners[320]={0}, wheel[160]={0},
	twowheels[320]={0}, plates[160]={0};

	//Detect Upper Corners
	Corners(img, corner, twocorners);

	//Detect Plate
	Plate(img, plates);

	//Detect Wheels
	Wheels(img, wheel, twowheels);

	//Group Cascades
	GroupParts(corner, plates, wheel, twocorners, twowheels, points, option);
}

//---------------------------------------------------------------------
//		Corners.cpp:	Detect Upper Corners of the Car
//		Created by:		Luis Aranda Barjola
//		Start Date:		23/06/12
//		Modify Date:	27/07/12
//		University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Corners(Mat& img, int* corner, int* twocorners)
{
	//Variables Definition
	int i=0, j=0, k=0, m=0, aux=0, media=0;
	vector<Rect> cars;
	Point vert1, vert2;

	//Load Haar Cascade Name
	String CascadeName = "C:/OpenCV2.2/Images/Cascades/esqcascade.xml";

	//Cascade Definition
	CascadeClassifier Cascade;

	//Load the Cascade
	Cascade.load(CascadeName);
```

```
//Create a Gray and Smaller Image
Mat gray, smallImg(cvRound(img.rows*2/3), cvRound(img.cols*8/10),
CV_8UC1);

//Compares the Image with the Cascade
Cascade.detectMultiScale(img, cars, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE,
Size(40, 40));

//Plot Rectangles for Every Car Detected
for(vector<Rect>::const_iterator r = cars.begin(); r != cars.end(); r++,
i++){
	media=0;
	for(j=0; j<=r->width*img.step/img.cols*8/10; j++){
		media = media + img.data[(r->y + r->height)*img.step+(r-
		>x)*img.step/img.cols*8/10+j];
	}
	media = media/(j-1);
	if(media < 160){
		//Rectangle
		vert1.x = r->x;
		vert1.y = r->y;
		vert2.x = (r->x + r->width);
		vert2.y = (r->y + r->height);

		//Improve Accuracy, Don't Save Rectangles Bigger than the Car
		if(r->width <= SIZE2){
			//Save Info in corner
			corner[k] = vert1.x;
			corner[k+1] = vert1.y;
			corner[k+2] = vert2.x;
			corner[k+3] = vert2.y;
			k = k + 4;
		}
	}
}
//Try Join Pairs of Corners
m = (k/4)-1;
k = 0;
for(i=0; i<m; i++){
	for(j=i+1; j<m+1; j++){
		aux = corner[4*i]-corner[4*j];
		//Corner "i" is RIGHT CORNER
		if(aux > 0){
			//Corner Distance Between SIZE1 & SIZE2
			if((aux > SIZE1)&&(aux < SIZE2)){
				//Corners Must Be in the Same Row (Approx.)
				if(abs(corner[4*i+1]-corner[4*j+1]) <
				cvRound((corner[4*i+3]-corner[4*i])/2)){
					//Left Corner
					twocorners[k] = corner[4*j];
					twocorners[k+1] = corner[4*j+1];
					twocorners[k+2] = corner[4*j+2];
					twocorners[k+3] = corner[4*j+3];
					//Right Corner
					twocorners[k+4] = corner[4*i];
					twocorners[k+5] = corner[4*i+1];
					twocorners[k+6] = corner[4*i+2];
					twocorners[k+7] = corner[4*i+3];
					k = k + 8;
				}
```

```
                        }
                    }
                    //Corner "j" is RIGHT CORNER
                    else{
                        //Corner Distance Between SIZE1 & SIZE2
                        if((abs(aux) > SIZE1)&&(abs(aux) < SIZE2)){
                            //Corners Must Be in the Same Row (Approx.)
                            if(abs(corner[4*i+1]-corner[4*j+1]) <
                            cvRound((corner[4*i+3]-corner[4*i])/2)){
                                //Left Corner
                                twocorners[k] = corner[4*i];
                                twocorners[k+1] = corner[4*i+1];
                                twocorners[k+2] = corner[4*i+2];
                                twocorners[k+3] = corner[4*i+3];
                                //Right Corner
                                twocorners[k+4] = corner[4*j];
                                twocorners[k+5] = corner[4*j+1];
                                twocorners[k+6] = corner[4*j+2];
                                twocorners[k+7] = corner[4*j+3];
                                k = k + 8;
                            }
                        }
                    }
                }
        }
}

//---------------------------------------------------------------------
//            Plate.cpp:          Detect Plate of the Car
//            Created by:         Luis Aranda Barjola
//            Start Date:         23/06/12
//            Modify Date:  27/07/12
//            University Carlos III de Madrid (Spain)
//---------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Plate(Mat& img, int* plates)
{
       //Variables Definition
       int i=0, j=0, k=0, media=0;
       vector<Rect> cars;
       Point vert1, vert2;

       //Load Haar Cascade Name
       String CascadeName = "C:/OpenCV2.2/Images/Cascades/matrcascade.xml";

       //Cascade Definition
       CascadeClassifier Cascade;

       //Load the Cascade
       Cascade.load(CascadeName);

       //Create a Gray and Smaller Image
       Mat gray, smallImg(cvRound(img.rows*2/3), cvRound(img.cols*8/10),
       CV_8UC1);

       //Compares the Image with the Cascade
       Cascade.detectMultiScale(img, cars, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE,
       Size(40, 40));
```

```
	//Plot Rectangles for Every Car Detected
	for(vector<Rect>::const_iterator r = cars.begin(); r != cars.end(); r++,
	i++){
		media=0;
		for(j=0; j<=r->width*img.step/img.cols*8/10; j++){
			media = media + img.data[(r->y + r->height)*img.step+(r-
			>x)*img.step/img.cols*8/10+j];
		}
		media = media/(j-1);
		if(media < 160){
			//Rectangle
			vert1.x = r->x;
			vert1.y = r->y;
			vert2.x = (r->x + r->width);
			vert2.y = (r->y + r->height);

			//Improve Accuracy, Don't Draw Rectangles Bigger than the Car
			if(r->width <= SIZE2){
				//Save Info in plates
				plates[k] = vert1.x;
				plates[k+1] = vert1.y;
				plates[k+2] = vert2.x;
				plates[k+3] = vert2.y;
				k = k + 4;
			}
		}
	}
}

//-------------------------------------------------------------------------
//		Wheels.cpp:		Detect Wheels of the Car
//		Created by:		Luis Aranda Barjola
//		Start Date:		23/06/12
//		Modify Date:	27/07/12
//		University Carlos III de Madrid (Spain)
//-------------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Wheels(Mat& img, int* wheel, int* twowheels)
{
	//Variables Definition
	int i=0, j=0, k=0, m=0, aux=0, media=0;
	vector<Rect> cars;
	Point vert1, vert2;

	//Load Haar Cascade Name
	String CascadeName = "C:/OpenCV2.2/Images/Cascades/ruedacascade.xml";

	//Cascade Definition
	CascadeClassifier Cascade;

	//Load the Cascade
	Cascade.load(CascadeName);

	//Create a Gray and Smaller Image
	Mat gray, smallImg(cvRound(img.rows*2/3), cvRound(img.cols*8/10),
	CV_8UC1);
```

```
//Compares the Image with the Cascade
Cascade.detectMultiScale(img, cars, 1.1, 2, 0|CV_HAAR_SCALE_IMAGE,
Size(40, 40));

//Plot Rectangles for Every Car Detected
for(vector<Rect>::const_iterator r = cars.begin(); r != cars.end(); r++,
i++){
	media=0;
	for(j=0; j<=r->width*img.step/img.cols*8/10; j++){
		media = media + img.data[(r->y + r->height)*img.step+(r-
		>x)*img.step/img.cols*8/10+j];
	}
	media = media / (j-1);
	if(media < 160){
		//Rectangle
		vert1.x = r->x;
		vert1.y = r->y;
		vert2.x = (r->x + r->width);
		vert2.y = (r->y + r->height);

		//Improve Accuracy, Don't Draw Rectangles Bigger than the Car
		if(r->width <= SIZE2){
			//Save Info in wheel
			wheel[k] = vert1.x;
			wheel[k+1] = vert1.y;
			wheel[k+2] = vert2.x;
			wheel[k+3] = vert2.y;
			k = k + 4;
		}
	}
}
//Try Join Pairs of Wheels
m = (k/4)-1;
k = 0;
for(i=0; i<m; i++){
	for(j=i+1; j<m+1; j++){
		aux = wheel[4*i]-wheel[4*j];
		//Wheel "i" is RIGHT WHEEL
		if(aux > 0){
			//Wheel Distance Between SIZE1 & SIZE2
			if((aux > SIZE1)&&(aux < SIZE2)){
				//Wheels Must Be in the Same Row (Approx.)
				if(abs(wheel[4*i+1]-wheel[4*j+1]) <
				cvRound((wheel[4*i+3]-wheel[4*i])/2)){
					//Left Wheel
					twowheels[k] = wheel[4*j];
					twowheels[k+1] = wheel[4*j+1];
					twowheels[k+2] = wheel[4*j+2];
					twowheels[k+3] = wheel[4*j+3];
					//Right Wheel
					twowheels[k+4] = wheel[4*i];
					twowheels[k+5] = wheel[4*i+1];
					twowheels[k+6] = wheel[4*i+2];
					twowheels[k+7] = wheel[4*i+3];
					k = k + 8;
				}
			}
		}
```

```
                        //Wheel "j" is RIGHT WHEEL
                        else{
                                //Wheel Distance Between SIZE1 & SIZE2
                                if((abs(aux) > SIZE1)&&(abs(aux) < SIZE2)){
                                        //Wheels Must Be in the Same Row (Approx.)
                                        if(abs(wheel[4*i+1]-wheel[4*j+1]) <
                                        cvRound((wheel[4*i+3]-wheel[4*i])/2)){
                                                //Left Wheel
                                                twowheels[k] = wheel[4*i];
                                                twowheels[k+1] = wheel[4*i+1];
                                                twowheels[k+2] = wheel[4*i+2];
                                                twowheels[k+3] = wheel[4*i+3];
                                                //Right Wheel
                                                twowheels[k+4] = wheel[4*j];
                                                twowheels[k+5] = wheel[4*j+1];
                                                twowheels[k+6] = wheel[4*j+2];
                                                twowheels[k+7] = wheel[4*j+3];
                                                k = k + 8;
                                        }
                                }
                        }
                }
        }
}

//-------------------------------------------------------------------
//              Distance.cpp: Calculate Distances in the Image
//              Created by:             Luis Aranda Barjola
//              Start Date:             28/07/12
//              Modify Date:  04/08/12
//              University Carlos III de Madrid (Spain)
//-------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

//Stereo Camera Parameters
struct StereoCamera{
        float Baseline; //m
        float DistanciaFocal;
        float CenterRow;
        float CenterCol;
};

void Distance(Mat& img, IplImage* DispMap, int* points, int* option, float* dist)
{
        //Variables Definition
        short val;
        short *porig16;
        unsigned char *p_imadisp;
        float u, v, X, Y, Z, D, f, fval1, fval2, fval3, fval4;
        double DispMin, DispMax;
        int i=0, j=0, k=0, k2=0, k3=0, cte=0, auxpoints[1000]={0},
        auxoption[7]={0};
        Point vert1, vert2;

        //Definition of Color
        const static Scalar colors[] = {CV_RGB(0,0,255)};
        Scalar color = colors[0];
```

```
//Camera Definition
struct StereoCamera Camera;
Camera.Baseline = 0.119915; //m
Camera.DistanciaFocal = 811.9104;
Camera.CenterRow = 247.1637;
Camera.CenterCol = 321.561;

//Calculate Min & Max Disparity
cvMinMaxLoc(DispMap, &DispMin, &DispMax);

//Camera Parameters
f = Camera.DistanciaFocal;
D = Camera.Baseline;

//Disparity
p_imadisp = (unsigned char*)DispMap->imageData;

//Calculate Distances
for(i=1; i<7; i++){
	if(option[i] != 0){
		for(j=0; j<option[i]; j++){
			//----------
			// OPTION 1
			//----------
			if(i == 1){
				//Left Corner -> Right-lower Point
				porig16 = (short*)(p_imadisp +
				points[20*j+3]*DispMap->widthStep);
				porig16 += points[20*j+2];
				val =* porig16;
				fval1 = val/RESOLUCION;

				//Right Corner -> Left-lower Point
				porig16 = (short*)(p_imadisp +
				points[20*j+7]*DispMap->widthStep);
				porig16 += points[20*j+4];
				val =* porig16;
				fval2 = val/RESOLUCION;

				//Left Wheel -> Right-upper Point
				porig16 = (short*)(p_imadisp +
				points[20*j+9]*DispMap->widthStep);
				porig16 += points[20*j+10];
				val =* porig16;
				fval3 = val/RESOLUCION;

				//Right Wheel -> Left-upper Point
				porig16 = (short*)(p_imadisp +
				points[20*j+13]*DispMap->widthStep);
				porig16 += points[20*j+12];
				val =* porig16;
				fval4 = val/RESOLUCION;

				//If Corner Distances are Equal
				if(fval1 == fval2){
					//If Wheels Distances are Equal ->
					Complete Car
					if(fval3 == fval4){
```

```
//---------
// CORNERS
//---------
//Left Corner
vert1.x = points[20*j];
vert1.y = points[20*j+1];
vert2.x = points[20*j+2];
vert2.y = points[20*j+3];
rectangle(img, vert1, vert2, color, 3, 8, 0);
//Right Corner
vert1.x = points[20*j+4];
vert1.y = points[20*j+5];
vert2.x = points[20*j+6];
vert2.y = points[20*j+7];
rectangle(img, vert1, vert2, color, 3, 8, 0);
//--------
// WHEELS
//--------
//Left Wheel
vert1.x = points[20*j+8];
vert1.y = points[20*j+9];
vert2.x = points[20*j+10];
vert2.y = points[20*j+11];
rectangle(img, vert1, vert2, color, 3, 8, 0);
//Right Wheel
vert1.x = points[20*j+12];
vert1.y = points[20*j+13];
vert2.x = points[20*j+14];
vert2.y = points[20*j+15];
rectangle(img, vert1, vert2, color, 3, 8, 0);
//Save Info
for(k2=0; k2<16; k2++){
	auxpoints[k2+k3] = points[20*j+k2];
}
k3 = k3 + 20;
auxoption[1]++;
//-------
// PLATE
//-------
//If Plate
if(points[20*j+16] != 0){
	vert1.x = points[20*j+16];
	vert1.y = points[20*j+17];
	vert2.x = points[20*j+18];
	vert2.y = points[20*j+19];
	rectangle(img, vert1, vert2, color, 3, 8, 0);
}

//--------------------
// CALCULATE DISTANCE
//--------------------
//Distance Z
Z = f*D/fval1;
if(Z <= DISTMAX){
	//Distance X-Y
	u = points[20*j] - Camera.CenterCol;
	v = points[20*j+1] - Camera.CenterRow;
	X = Z*u/f;
	Y = Z*v/f;
```

```
		//If First Time
		if(dist[0] == 0){
			dist[0] = Z;
			dist[1] = X;
			dist[2] = Y;
			k = k + 3;
		}
		//If not First Time
		else{
			//If Equal Distance -> Don't Save
			if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
				dist[k] = Z;
				dist[k+1] = X;
				dist[k+2] = Y;
				k = k + 3;
			}
		}
}
}
}
}
//----------
// OPTION 2
//----------
if(i == 2){
		//Calculate Constant
		cte = 20*option[1];
		//Left Corner -> Right-lower Point
		porig16 = (short*)(p_imadisp + points[cte+16*j+3]*DispMap->widthStep);
		porig16 += points[cte+16*j+2];
		val =* porig16;
		fval1 = val/RESOLUCION;

		//Right Corner -> Left-lower Point
		porig16 = (short*)(p_imadisp + points[cte+16*j+7]*DispMap->widthStep);
		porig16 += points[cte+16*j+4];
		val =* porig16;
		fval2 = val/RESOLUCION;

		//Left Wheel -> Right-upper Point
		porig16 = (short*)(p_imadisp + points[cte+16*j+9]*DispMap->widthStep);
		porig16 += points[cte+16*j+10];
		val =* porig16;
		fval3 = val/RESOLUCION;

		//If Three Distances Equal -> OK
		if((fval1 == fval2)&&(fval1 == fval3)){
		//---------
		// CORNERS
		//---------
		//Left Corner
		vert1.x = points[cte+16*j];
		vert1.y = points[cte+16*j+1];
		vert2.x = points[cte+16*j+2];
		vert2.y = points[cte+16*j+3];
		rectangle(img, vert1, vert2, color, 3, 8, 0);
		//Right Corner
		vert1.x = points[cte+16*j+4];
		vert1.y = points[cte+16*j+5];
		vert2.x = points[cte+16*j+6];
```

```
        vert2.y = points[cte+16*j+7];
        rectangle(img, vert1, vert2, color, 3, 8, 0);
        //-------
        // WHEEL
        //-------
        //Left Wheel
        vert1.x = points[cte+16*j+8];
        vert1.y = points[cte+16*j+9];
        vert2.x = points[cte+16*j+10];
        vert2.y = points[cte+16*j+11];
        rectangle(img, vert1, vert2, color, 3, 8, 0);
        //Save Info
        for(k2=0; k2<12; k2++){
                auxpoints[k2+k3] = points[20*j+k2];
        }
        k3 = k3 + 16;
        auxoption[2]++;
        //-------
        // PLATE
        //-------
        //If Plate
        if(points[cte+16*j+12] != 0){
                vert1.x = points[cte+16*j+12];
                vert1.y = points[cte+16*j+13];
                vert2.x = points[cte+16*j+14];
                vert2.y = points[cte+16*j+15];
                rectangle(img, vert1, vert2, color, 3, 8, 0);
        }

        //--------------------
        // CALCULATE DISTANCE
        //--------------------
        //Distance Z
        Z = f*D/fval1;
        if(Z <= DISTMAX){
                //Distance X-Y
                u = points[cte+16*j] - Camera.CenterCol;
                v = points[cte+16*j+1] - Camera.CenterRow;
                X = Z*u/f;
                Y = Z*v/f;
                //If First Time
                if(dist[0] == 0){
                        dist[0] = Z;
                        dist[1] = X;
                        dist[2] = Y;
                        k = k + 3;
                }
                //If not First Time
                else{
                        //If Equal Distance -> Don't Save
                        if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
                                dist[k] = Z;
                                dist[k+1] = X;
                                dist[k+2] = Y;
                                k = k + 3;
                        }
                }
        }
}
}
```

```
//----------
// OPTION 3
//----------
if(i == 3){
	//Calculate Constant
	cte = 20*option[1] + 16*option[2];
	//Left Corner -> Right-lower Point
	porig16 = (short*)(p_imadisp + points[cte+16*j+3]*DispMap->widthStep);
	porig16 += points[cte+16*j+2];
	val =* porig16;
	fval1 = val/RESOLUCION;

	//Right Corner -> Left-lower Point
	porig16 = (short*)(p_imadisp + points[cte+16*j+7]*DispMap->widthStep);
	porig16 += points[cte+16*j+4];
	val =* porig16;
	fval2 = val/RESOLUCION;

	//Right Wheel -> Left-upper Point
	porig16 = (short*)(p_imadisp + points[cte+16*j+9]*DispMap->widthStep);
	porig16 += points[cte+16*j+8];
	val =* porig16;
	fval3 = val/RESOLUCION;

	//If Three Distances Equal -> OK
	if((fval1 == fval2)&&(fval1 == fval3)){
		//---------
		// CORNERS
		//---------
		//Left Corner
		vert1.x = points[cte+16*j];
		vert1.y = points[cte+16*j+1];
		vert2.x = points[cte+16*j+2];
		vert2.y = points[cte+16*j+3];
		rectangle(img, vert1, vert2, color, 3, 8, 0);
		//Right Corner
		vert1.x = points[cte+16*j+4];
		vert1.y = points[cte+16*j+5];
		vert2.x = points[cte+16*j+6];
		vert2.y = points[cte+16*j+7];
		rectangle(img, vert1, vert2, color, 3, 8, 0);
		//-------
		// WHEEL
		//-------
		//Right Wheel
		vert1.x = points[cte+16*j+8];
		vert1.y = points[cte+16*j+9];
		vert2.x = points[cte+16*j+10];
		vert2.y = points[cte+16*j+11];
		rectangle(img, vert1, vert2, color, 3, 8, 0);
		//Save Info
		for(k2=0; k2<12; k2++){
			auxpoints[k2+k3] = points[20*j+k2];
		}
		k3 = k3 + 16;
		auxoption[3]++;
		//-------
		// PLATE
		//-------
		//If Plate
```

```
                if(points[cte+16*j+12] != 0){
                        vert1.x = points[cte+16*j+12];
                        vert1.y = points[cte+16*j+13];
                        vert2.x = points[cte+16*j+14];
                        vert2.y = points[cte+16*j+15];
                        rectangle(img, vert1, vert2, color, 3, 8, 0);
                }

                //--------------------
                // CALCULATE DISTANCE
                //--------------------
                //Distance Z
                Z = f*D/fval1;
                if(Z <= DISTMAX){
                        //Distance X-Y
                        u = points[cte+16*j] - Camera.CenterCol;
                        v = points[cte+16*j+1] - Camera.CenterRow;
                        X = Z*u/f;
                        Y = Z*v/f;
                        //If First Time
                        if(dist[0] == 0){
                                dist[0] = Z;
                                dist[1] = X;
                                dist[2] = Y;
                                k = k + 3;
                        }
                        //If not First Time
                        else{
                        //If Equal Distance -> Don't Save
                        if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
                                dist[k] = Z;
                                dist[k+1] = X;
                                dist[k+2] = Y;
                                k = k + 3;
                        }
                        }
                }
        }
}
//----------
// OPTION 4
//----------
if(i == 4){
//Calculate Constant
cte = 20*option[1] + 16*option[2] + 16*option[3];
//Left Corner -> Right-lower Point
porig16 = (short*)(p_imadisp + points[cte+16*j+3]*DispMap->widthStep);
porig16 += points[cte+16*j+2];
val =* porig16;
fval1 = val/RESOLUCION;

//Left Wheel -> Right-upper Point
porig16 = (short*)(p_imadisp + points[cte+16*j+5]*DispMap->widthStep);
porig16 += points[cte+16*j+6];
val =* porig16;
fval2 = val/RESOLUCION;

//Right Wheel -> Left-upper Point
porig16 = (short*)(p_imadisp + points[cte+16*j+9]*DispMap->widthStep);
porig16 += points[cte+16*j+8];
```

```
val =* porig16;
fval3 = val/RESOLUCION;

//If Three Distances Equal -> OK
if((fval1 == fval2)&&(fval1 == fval3)){
	//--------
	// CORNER
	//--------
	//Left Corner
	vert1.x = points[cte+16*j];
	vert1.y = points[cte+16*j+1];
	vert2.x = points[cte+16*j+2];
	vert2.y = points[cte+16*j+3];
	rectangle(img, vert1, vert2, color, 3, 8, 0);
	//--------
	// WHEELS
	//--------
	//Left Wheel
	vert1.x = points[cte+16*j+4];
	vert1.y = points[cte+16*j+5];
	vert2.x = points[cte+16*j+6];
	vert2.y = points[cte+16*j+7];
	rectangle(img, vert1, vert2, color, 3, 8, 0);
	//Right Wheel
	vert1.x = points[cte+16*j+8];
	vert1.y = points[cte+16*j+9];
	vert2.x = points[cte+16*j+10];
	vert2.y = points[cte+16*j+11];
	rectangle(img, vert1, vert2, color, 3, 8, 0);
	//Save Info
	for(k2=0; k2<12; k2++){
		auxpoints[k2+k3] = points[20*j+k2];
	}
	k3 = k3 + 16;
	auxoption[4]++;
	//-------
	// PLATE
	//-------
	//If Plate
	if(points[cte+16*j+12] != 0){
		vert1.x = points[cte+16*j+12];
		vert1.y = points[cte+16*j+13];
		vert2.x = points[cte+16*j+14];
		vert2.y = points[cte+16*j+15];
		rectangle(img, vert1, vert2, color, 3, 8, 0);
	}

	//--------------------
	// CALCULATE DISTANCE
	//--------------------
	//Distance Z
	Z = f*D/fval1;
	if(Z <= DISTMAX){
		//Distance X-Y
		u = points[cte+16*j] - Camera.CenterCol;
		v = points[cte+16*j+1] - Camera.CenterRow;
		X = Z*u/f;
		Y = Z*v/f;
```

```
                //If First Time
                if(dist[0] == 0){
                        dist[0] = Z;
                        dist[1] = X;
                        dist[2] = Y;
                        k = k + 3;
                }
                //If not First Time
                else{
                        //If Equal Distance -> Don't Save
                        if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
                                dist[k] = Z;
                                dist[k+1] = X;
                                dist[k+2] = Y;
                                k = k + 3;
                        }
                }
        }
}
}
//----------
// OPTION 5
//----------
if(i == 5){
        //Calculate Constant
        cte = 20*option[1] + 16*option[2] + 16*option[3] + 16*option[4];
        //Right Corner -> Left-lower Point
        porig16 = (short*)(p_imadisp + points[cte+16*j+3]*DispMap->widthStep);
        porig16 += points[cte+16*j];
        val =* porig16;
        fval1 = val/RESOLUCION;

        //Left Wheel -> Right-upper Point
        porig16 = (short*)(p_imadisp + points[cte+16*j+5]*DispMap->widthStep);
        porig16 += points[cte+16*j+6];
        val =* porig16;
        fval2 = val/RESOLUCION;

        //Right Wheel -> Left-upper Point
        porig16 = (short*)(p_imadisp + points[cte+16*j+9]*DispMap->widthStep);
        porig16 += points[cte+16*j+8];
        val =* porig16;
        fval3 = val/RESOLUCION;

        //If Three Distances Equal -> OK
        if((fval1 == fval2)&&(fval1 == fval3)){
                //--------
                // CORNER
                //--------
                //Right Corner
                vert1.x = points[cte+16*j];
                vert1.y = points[cte+16*j+1];
                vert2.x = points[cte+16*j+2];
                vert2.y = points[cte+16*j+3];
                rectangle(img, vert1, vert2, color, 3, 8, 0);
                //--------
                // WHEELS
                //--------
                //Left Wheel
                vert1.x = points[cte+16*j+4];
```

```
                vert1.y = points[cte+16*j+5];
                vert2.x = points[cte+16*j+6];
                vert2.y = points[cte+16*j+7];
                rectangle(img, vert1, vert2, color, 3, 8, 0);
                //Right Wheel
                vert1.x = points[cte+16*j+8];
                vert1.y = points[cte+16*j+9];
                vert2.x = points[cte+16*j+10];
                vert2.y = points[cte+16*j+11];
                rectangle(img, vert1, vert2, color, 3, 8, 0);
                //Save Info
                for(k2=0; k2<12; k2++){
                        auxpoints[k2+k3] = points[20*j+k2];
                }
                k3 = k3 + 16;
                auxoption[5]++;
                //-------
                // PLATE
                //-------
                //If Plate
                if(points[cte+16*j+12] != 0){
                        vert1.x = points[cte+16*j+12];
                        vert1.y = points[cte+16*j+13];
                        vert2.x = points[cte+16*j+14];
                        vert2.y = points[cte+16*j+15];
                        rectangle(img, vert1, vert2, color, 3, 8, 0);
                }

                //--------------------
                // CALCULATE DISTANCE
                //--------------------
                //Distance Z
                Z = f*D/fval1;
                if(Z <= DISTMAX){
                        //Distance X-Y
                        u = points[cte+16*j+4] - Camera.CenterCol;
                        v = points[cte+16*j+1] - Camera.CenterRow;
                        X = Z*u/f;
                        Y = Z*v/f;
                        //If First Time
                        if(dist[0] == 0){
                                dist[0] = Z;
                                dist[1] = X;
                                dist[2] = Y;
                                k = k + 3;
                        }
                        //If not First Time
                        else{
                        //If Equal Distance -> Don't Save
                        if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
                                dist[k] = Z;
                                dist[k+1] = X;
                                dist[k+2] = Y;
                                k = k + 3;
                        }
                        }
                }
        }
}
```

```
//----------
// OPTION 6
//----------
if(i == 6){
//Calculate Constant
cte = 20*option[1] + 16*option[2] + 16*option[3] + 16*option[4] + 16*option[5];
       //Corner
       porig16 = (short*)(p_imadisp + points[cte+12*j+3]*DispMap->widthStep);
       porig16 += cvRound((points[cte+12*j]+points[cte+12*j+2])/2);
       val =* porig16;
       fval1 = val/RESOLUCION;

       //Wheel
       porig16 = (short*)(p_imadisp + points[cte+12*j+5]*DispMap->widthStep);
       porig16 += cvRound((points[cte+12*j+4]+points[cte+12*j+6])/2);
       val =* porig16;
       fval2 = val/RESOLUCION;

       //Plate
       porig16 = (short*)(p_imadisp + points[cte+12*j+9]*DispMap->widthStep);
       porig16 += points[cte+12*j+8];
       val =* porig16;
       fval3 = val/RESOLUCION;

       //If Three Distances Equal -> OK
       if((fval1 == fval2)&&(fval1 == fval3)){
       //--------
       // CORNER
       //--------
       //Corner
       vert1.x = points[cte+12*j];
       vert1.y = points[cte+12*j+1];
       vert2.x = points[cte+12*j+2];
       vert2.y = points[cte+12*j+3];
       rectangle(img, vert1, vert2, color, 3, 8, 0);
       //-------
       // WHEEL
       //-------
       //Wheel
       vert1.x = points[cte+12*j+4];
       vert1.y = points[cte+12*j+5];
       vert2.x = points[cte+12*j+6];
       vert2.y = points[cte+12*j+7];
       rectangle(img, vert1, vert2, color, 3, 8, 0);
       //-------
       // PLATE
       //-------
       //Plate
       vert1.x = points[cte+12*j+8];
       vert1.y = points[cte+12*j+9];
       vert2.x = points[cte+12*j+10];
       vert2.y = points[cte+12*j+11];
       rectangle(img, vert1, vert2, color, 3, 8, 0);
       //Save Info
       for(k2=0; k2<12; k2++){
              auxpoints[k2+k3] = points[20*j+k2];
       }
       k3 = k3 + 12;
       auxoption[6]++;
```

```
        //--------------------
        // CALCULATE DISTANCE
        //--------------------
        //Distance Z
        Z = f*D/fval1;
        if(Z <= DISTMAX){
                //Distance X-Y
                u = points[cte+12*j] - Camera.CenterCol;
                v = points[cte+12*j+1] - Camera.CenterRow;
                X = Z*u/f;
                Y = Z*v/f;
                //If First Time
                if(dist[0] == 0){
                        dist[0] = Z;
                        dist[1] = X;
                        dist[2] = Y;
                        k = k + 3;
                }
                //If not First Time
                else{
                        //If Equal Distance -> Don't Save
                        if((dist[k-3] != Z)||(dist[k-2] != X)||(dist[k-1] != Y)){
                                dist[k] = Z;
                                dist[k+1] = X;
                                dist[k+2] = Y;
                                k = k + 3;
                        }
                }
        }
}
}
}
}
}
//Show Results
if(dist[0] != 0){
        printf(" -----------------\n");
        printf(" |   DISTANCES   |\n");
        printf(" -----------------\n");
        i=0; j=1;
        while(dist[i] != 0){
                printf(" ----- CAR %d -----\n", j);
                printf(" | X: %fm |\n", dist[i+1]);
                printf(" | Y: %fm |\n", dist[i+2]);
                printf(" | Z: %fm |\n", dist[i]);
                i = i + 3;
                j++;
        }
        printf(" -----------------\n");
}

//Update Points & Option
for(i=0; i<1000; i++){
        points[i] = auxpoints[i];
}
for(i=0; i<7; i++){
        option[i] = auxoption[i];
}
}
```

```
//-------------------------------------------------------------------
//            Searching.cpp:             Search Detected Cars
//            Created by:                Luis Aranda Barjola
//            Start Date:                02/08/12
//            Modify Date:        02/08/12
//            University Carlos III de Madrid (Spain)
//-------------------------------------------------------------------
#include "StdAfx.h"
#include "Main_Header.h"

void Searching(Mat& img, char* Imagename, int* simi, int* coord, int* points,
int* option)
{
	//Variables Definition
	int sim=0, i=0, j=0, k=0, k2=0, k3=0, cte=0, dif=0, aux=0, auxsim[15]={0},
	auxcoord[60]={0};
	IplImage *img2;

	//Definition of Colors
	const static Scalar colors[] = {CV_RGB(0,0,255)};
	Scalar color = colors[0];

	//Load Original Image
	IplImage *img1 = cvLoadImage(Imagename, CV_LOAD_IMAGE_GRAYSCALE);

	//Calculate Vertical Rectangle ROI Coordinates
	//------------
	// First Time
	//------------
	if(simi[0] == 0){
		for(i=1; i<7; i++){
			if(option[i] != 0){
			for(j=0; j<option[i]; j++){
			//----------
			// OPTION 1
			//----------
			if(i == 1){
			coord[k] = cvRound((points[20*j]+points[20*j+6])/2);
			coord[k+1] = points[20*j+1];
			coord[k+2] = points[20*j+11];
			k = k + 3;
			}
			//----------
			// OPTION 2
			//----------
			if(i == 2){
			cte = 20*option[1];
			coord[k] = cvRound((points[cte+16*j]+points[cte+16*j+6])/2);
			coord[k+1] = points[cte+16*j+1];
			coord[k+2] = points[cte+16*j+11];
			k = k + 3;
			}
			//----------
			// OPTION 3
			//----------
			if(i == 3){
			cte = 20*option[1] + 16*option[2];
			coord[k] = cvRound((points[cte+16*j]+points[cte+16*j+6])/2);
			coord[k+1] = points[cte+16*j+1];
			coord[k+2] = points[cte+16*j+11];
```

```
                    k = k + 3;
                    }
                    //----------
                    // OPTION 4
                    //----------
                    if(i == 4){
                    cte = 20*option[1] + 16*option[2] + 16*option[3];
                    coord[k] = cvRound((points[cte+16*j+4]+points[cte+16*j+10])/2);
                    coord[k+1] = points[cte+16*j+1];
                    coord[k+2] = points[cte+16*j+7];
                    k = k + 3;
                    }
                    //----------
                    // OPTION 5
                    //----------
                    if(i == 5){
                    cte = 20*option[1] + 16*option[2] + 16*option[3] + 16*option[4];
                    coord[k] = cvRound((points[cte+16*j+4]+points[cte+16*j+10])/2);
                    coord[k+1] = points[cte+16*j+1];
                    coord[k+2] = points[cte+16*j+7];
                    k = k + 3;
                    }
                    //----------
                    // OPTION 6
                    //----------
                    if(i == 6){
                    cte = 20*option[1] + 16*option[2] + 16*option[3] +
                    16*option[4] + 16*option[5];
                    coord[k] = cvRound((points[cte+12*j+8]+points[cte+12*j+10])/2);
                    coord[k+1] = points[cte+12*j+1];
                    coord[k+2] = points[cte+12*j+7];
                    k = k + 3;
                    }
            }
        }
}

//Reset Counters
i=0; k=0;

//Calculate Interquartile Range
while(coord[k] != 0){
        //Draw Vertical Lines
        CvPoint p1 = cvPoint(coord[k], coord[k+1]);
        CvPoint p2 = cvPoint(coord[k], coord[k+2]);
        line(img, p1, p2, color, 3, 8, 0);

        //Create ROI
        cvSetImageROI(img1, cvRect(coord[k]-3, coord[k+1], 6, coord[k+2]-
        coord[k+1]));
        img2 = cvCreateImage(cvGetSize(img1), img1->depth, img1->nChannels);

        //Copy Subimage
        cvCopy(img1, img2, NULL);

        //Call Histograms
        Histograms(img2, sim);

        //Save Similarity
        simi[i] = sim;
```

```
		//Increment Counters
		i++;
		k = k + 3;
}
//Free ROI & Images
cvResetImageROI(img1);
cvReleaseImage(&img2);
cvReleaseImage(&img1);
}
//-----------------
// Following Times
//-----------------
else{
		while(coord[k] != 0){
			for(i=0; i<10; i++){
				//Left Scanning
				if(i <= 5){
				//Create ROI
				cvSetImageROI(img1, cvRect(coord[k]-i, coord[k+1], 6,
				coord[k+2]-coord[k+1]));
				}
				//Right Scanning
				else{
				//Create ROI
				cvSetImageROI(img1, cvRect(coord[k]+i-5, coord[k+1], 6,
				coord[k+2]-coord[k+1]));
				}

				//Create Another Image
				img2 = cvCreateImage(cvGetSize(img1), img1->depth, img1-
				>nChannels);

				//Copy Subimage
				cvCopy(img1, img2, NULL);

				//Call Histograms
				Histograms(img2, sim);

				//Obtain More Similar
				while(simi[j] != 0){
					dif = abs(simi[j]-sim);
					//First Time
					if((aux == 0)&&(dif != 0)){
						aux = dif;
						auxsim[k2] = sim;
						if(i <= 5){
							auxcoord[k3] = coord[k]-i;
						}
						else{
							auxcoord[k3] = coord[k]+i-5;
						}
						auxcoord[k3+1] = coord[k+1];
						auxcoord[k3+2] = coord[k+2];
					}
```

```
                        //Following Times
                        else{
                                if(dif < aux){
                                        aux = dif;
                                        auxsim[k2] = sim;
                                        if(i <= 5){
                                                auxcoord[k3] = coord[k]-i;
                                        }
                                        else{
                                                auxcoord[k3] = coord[k]+i-5;
                                        }
                                        auxcoord[k3+1] = coord[k+1];
                                        auxcoord[k3+2] = coord[k+2];
                                }
                        }
                        j++;
                }
                //Reset Counter
                j=0;
        }

        //Draw Vertical Lines
        CvPoint p1 = cvPoint(auxcoord[k3], auxcoord[k3+1]);
        CvPoint p2 = cvPoint(auxcoord[k3], auxcoord[k3+2]);
        line(img, p1, p2, color, 2, 8, 0);

        //Increment Counters
        k = k + 3;
        k2++;
        k3 = k3 + 3;
    }
    //Free ROI & Images
    cvResetImageROI(img1);
    cvReleaseImage(&img2);
    cvReleaseImage(&img1);

    //Save New Similarity & New Coordinates
    for(i=0; i<15; i++){
        simi[i] = auxsim[i];
    }
    for(i=0; i<60; i++){
        coord[i] = auxcoord[i];
    }
}
}
```